Pick Your Picker With Color Picker 2.0 • Media Capture Using the Sequence Grabber

develop

The Apple Technical Journal

Issue 19 September 1994

Building an OpenDoc Part Handler

Tuning PowerPC Memory Usage

Designing for the Power Macintosh

Adding QuickDraw GX Printing to QuickDraw Applications

Making the Most of QuickDraw GX Bitmaps

Implementing Inheritance in Scripts

10.00

develop

EDITORIAL STAFF

ART & PRODUCTION

ISSN #1047-0735.

Printed on recycled paper

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December. *develop* articles and code have been reviewed for robustness by Apple engineers.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. The Bookmark CD contains a subset of the materials on the monthly *Developer CD Series*, which is available from APDA. Included on the CD are this issue and all back issues of *develop* along with the code that the articles describe. The *develop* code is updated when necessary, so always use the most recent CD. The CD also contains Technical Notes, sample code, and other useful documentation and tools (these contents are subject to change). Software and documentation referred to as being on this issue's CD are located on either the Bookmark CD or the Reference Library or Tool Chest edition of the *Developer CD Series*.

The *develop* issues and code are also available on AppleLink and via anonymous ftp at ftp.apple.com.

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something in parentheses like "(Memory 13)," this indicates the category and number of the Note on this issue's CD.

E-mail addresses. Most e-mail addresses mentioned in *develop* are AppleLink addresses; to convert one of these to an Internet address, append "@applelink.apple.com" to it. For example, DEVELOP on AppleLink becomes develop@applelink.apple.com on the Internet. To convert a NewtonMail address to an Internet address, append "@online.apple.com" to it.

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)974-6395. Or write to Caroline Rose at the above address.

Subscriptions. Subscribe to *develop* through APDA (see below) or use the subscription card in this issue. For subscription changes or queries, call 1-800-877-5548 in the U.S. or (815)734-1116 outside the U.S., or write to AppleLink DEV.SUBS, Internet dev.subs@applelink.apple.com, or *develop*, P.O. Box 531, Mount Morris, IL 61054-7858.

Back issues. Printed back issues are available for $13 each in the U.S. or $20 outside the U.S. To order, call 1-800-877-5548 in the U.S. or (815)734-1116 outside the U.S., or write to AppleLink DEV.SUBS, Internet dev.subs@applelink.apple.com, or *develop*, P.O. Box 531, Mount Morris, IL 61054-7858.

APDA. To order products from APDA or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. Order electronically at AppleLink APDA, Internet apda@applelink.apple.com, CompuServe 76666,2405, or America Online APDAorder. Or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

Around the time I was faced with writing this editorial, I had just attended the celebration of my friend Mrs. Robertson's 100th birthday, and my 85-year-old father had flown over from Florida to celebrate it with us. With the subject of longevity on my mind, I got to thinking about how it relates to *develop*.

develop's goal is to provide you with articles and code that will have a long life — that can live in your applications happily and compatibly even as new Macintosh systems are introduced. We do all we can to ensure this (at the risk of incurring the wrath of our authors, who may wonder why it takes so long to see something in print after it's submitted to *develop*). We'd rather an article "have legs" than be published prematurely and get you into trouble further down the line. We do our best to test the code and get our technical reviewers' opinions on whether a particular method is safe. This should be a primary concern of all developers, especially now in light of the whole new world of Power Macintosh systems.

Evidence that we're succeeding is that we still get requests to reprint articles as far back as Issue 2, and we often hear from readers who save every issue because they retain their usefulness. (Remember that, the next time you're thinning out your bookshelves!)

Our being an Apple publication gives us the distinct advantage of being able to have a thorough code review with future systems in mind, but at the same time it puts us in a unique position to have early articles on new Apple technology. So we also try to give you articles as soon as possible after the API for a new technology has frozen. And if we can, we give you a prerelease version of the new software along with the code on our CD. These articles may have somewhat shorter legs, but the bulk of the information should remain accurate for a very long time.

In the past we've given you early QuickDraw GX versions and articles; now that QuickDraw GX has shipped, the two articles on that subject in this issue are only the most recent in a long line. Also in this issue we're pleased to bring you our first article on OpenDoc, Apple's new cross-platform compound-document architecture, even though the final version of OpenDoc will not have shipped by the time you read this.

Further evidence that we're succeeding is that we've again won in the International Technical Publications Competition of the Society for Technical Communication, this time the highest award in our category. But nothing would please us more than to hear from you, the most important judges of all, on what we can do to make *develop* an even better publication; please let us know at AppleLink DEVELOP.

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) As a child, Caroline wrote a one-page newsletter about the goings-on in her neighborhood; it included news items, a gossip column, and a comic strip. Her readership was small, and the operation folded after one issue. She's happy that *develop* has lasted longer than that, because after various jobs at Tymshare, NeXT, and Apple as a programmer, writer, editor, and manager, she feels she's found her niche here. These days when Caroline's not at work she's likely to be sailing, swimming, jogging, dancing, gardening, or otherwise not being sedentary. She hopes to live 100 very active years.

LETTERS

MISSING GAME FOLDER

The column by Brigham Stevens on game development in *develop* Issue 17 refers to a Game Development folder on the Bookmark 17 CD. I was unable to locate the folder. Is it me, or was there a production glitch?

Also, is the folder the same one that was on the February Developer CD?

— Bob Boonstra

There was a production glitch. The folder was on Issue 16's Bookmark CD but was inadvertently removed from Issue 17's CD. We have since restored it (it's in the Tools & Applications folder). We also got the name wrong: it's Games, not Game Development.

And yes, it's the same folder as the one on the Developer CD. The Bookmark CD always contains a subset of the Developer CD Series.

Happy gaming!

— Dave Johnson

DEVELOP ON THE SMALL SCREEN

In Issue 17 you ask why so many applications lack common sense, and then you go on to list a number of annoyances caused by bad designs. I agree that the items mentioned could be better handled, but I have a similar complaint with *develop*!

develop, shipped on CD-ROM with Apple DocViewer, is a software product that's produced like a printed document. Things like double-columned pages are extremely difficult to read on a standard Macintosh 13-inch monitor — you have to scroll down the entire page as you read a single column and then scroll back up and repeat the process. While this orientation makes sense on a printed page, it really bites on my 13-inch monitor. Also, *develop* is nearly unreadable with its serif typefaces, italics, etc.

Right now you're probably shaking your head and saying that you don't have the money to produce two versions of *develop* — one for print and one for CD. If so, then you also know why so many applications lack common sense!

— Brooks Bell

Certainly time and money do enter into design decisions, even at Apple. But there are still a lot of cases where common sense could be followed without a big hit to the schedule or pocketbook.

Regarding your problem with viewing the CD version of develop *on a 13-inch screen, you might try DocViewer's "text" view (the icon in the tool bar that looks like a sheet of paper with writing, just to the left of the scaling pop-up menu). This view gets rid of all special formatting such as double columns. You can still look at illustrations, by clicking the Open button next to the figure caption.*

In text view you can change the font size and type of any structural part of the document, using the Format command in the Edit menu. For example, you can choose Body in the Format dialog and change the font of all the body text. (In the normal view, the scaling pop-up menu can be used to magnify everything.)

Thanks for writing and giving us the opportunity to provide these tips. We welcome all gripes!

— Caroline Rose

WE'RE DYING TO HEAR FROM YOU

We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014 (AppleLink CROSE or JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

QUICKDRAW GX MORPH TABLES

The Macintosh Q&A section of *develop* Issue 16 stated that there's no way to do an automatic checksum digit insertion using QuickDraw GX's glyph metamorphosis tables. I haven't read much about QuickDraw GX's 'mort' tables, but I do know finite state tables. It's quite feasible to build a 12-state table that will generate a checksum digit for a digit-string of any length. The key is to make the checksum be calculated dynamically instead of at the end. This reduces the required number of states to ten, plus beginning and end, which is well within the limits of QuickDraw GX.

— Mark Cogan
Developer Wannabe

You're quite right; you can indeed use morph tables to generate checksums. Other cyclic kinds of calculations are also possible; for example, a morph table could be set up to do pseudo-random selection of letterforms from a font that was designed with five variant forms of A-Z and a-z. When a letter is encountered, it would be replaced with its version from one of the sets. But regardless of the specific glyph, each time a glyph is processed the state advances and eventually loops back to the starting state. This is in general the template for how cyclic effects can be implemented with the QuickDraw GX morph tables.

— Dave Opstad
GX Line Layout Weenie

WATCH-CURSOR PUSH-UPS AND BEYOND

Regarding Dave Johnson's bio in Issue 16: He's not the only one who plays with the cursor while waiting for the Mac. Here are some other silly things to do:

• Try to fit the watch inside an empty horizontal scroll bar. It always overlaps one of the lines, either the top or bottom one, so if you move that single pixel that alternates between them, the watch seems to be slipping on its wrist belt.

• While installing the system, position the counting hand so that it's just touching a horizontal line. Don't overlap the line, and the hand will seem to be cut from its owner, instead of being a ghost unterminated hand.

— Javier Guerra G.

Dave is not alone in doing watch-cursor push-ups and pull-ups. My personal favorite activity here is trying to find a place where I can do both at once. I remember that the old Font/DA Mover has a spot between two buttons where you can do that nicely.

— Maarten Hazewinkel

Hot Dawg! I knew there were others out there doing the watch cursor thing. I've heard from a half dozen or so; it's a lot more common than I thought.

I remember doing that with the old Font/DA Mover, too. The progress bar during long Finder operations also worked well. (Nowadays we have movable modal progress windows, so the watch cursor is gone — the price of progress.)

Thanks for letting me know I'm not alone!

— Dave Johnson

UNABASHED PRAISE

develop is an inspiring magazine. The layout is clean yet warm and inviting. The articles are relevant and the authors are knowlegeable.

The on-line issues of *develop* are invaluable. This is the best on-line documentation I have seen, period. The articles look great — just like the magazine. And searching and setting filters is fast. Microsoft's CD comes with a lot of files, but most of it is old, irrelevant, and ugly to read.

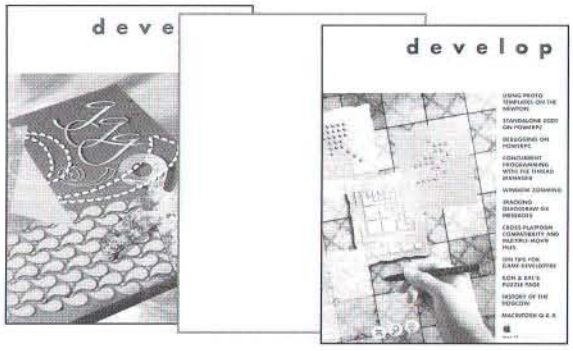
I sure appreciate your effort.

— Brent Foust

We can't thank you enough for taking the time to write. Letters like this keep us going, in more ways than one. We hope you're as happy with the recent changes to our layout; please let us know if not.

— Caroline Rose

The (w)hole collection



Are there issues of *develop* that have passed you by? If you'd like to complete your *develop* collection, full-color, bound copies are available. (Back issues are also on the *develop Bookmark* CD and the Reference Library edition of the *Developer CD Series*.)

To order printed back issues, send $13 per issue in the U.S. (or $20 outside the U.S.) to *develop* Back Issues, P.O. Box 531, Mount Morris, IL 61054-7858. Or call 1-800-877-5548 in the U.S. or (815)734-1116 elsewhere. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7 Compatibility; Debugging Declaration ROMs; Apple II Development Dynamo

Issue 2 C++ Objects; Object Pascal; Memory Manager; MacApp; How to Design an Object-Based Application; C++ Style Guide; The GS/OS Cache

Issue 3 ISO 9660 and High Sierra; A Mixed-Partition CD; Accessing CD Audio Tracks; Comm Toolbox; Macintosh Display Card 8•24 GC; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Scanning From ProDOS; Palette Manager Animation; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook: Constructing Network-Aware Applications

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Validating Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; The TextBox You've Always Wanted; Making Your Macintosh Sound Like an Echo Box; Simple Text Windows via the Terminal Manager; Tracks: A New Tool for Debugging Drivers

Issue 10 Apple Event Objects; PostScript Enhancements for the LaserWriter Font Utility; Drawing in GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 Asynchronous Routines; QuickTime and Component-Based Managers; Macintosh Debugging Revisited; Adventures in Color Printing; DeviceLoop

Issue 14 Writing Localizable Applications; 3-D Rotation Using a 2-D Input Device; Video Digitizing Under QuickTime; Making Better QuickTime Movies

Issue 15 QuickDraw GX (Getting Started; Printing Extensions; PostScript); Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Proto Templates on the Newton; Standalone Code on PowerPC; Debugging on PowerPC; Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Building an OpenDoc Part Handler

OpenDoc, Apple's compound-document architecture, brings users a new, more powerful metaphor for working with documents. Writing code to support OpenDoc is a lot like writing a normal application. This article gives an overview of what's involved in writing OpenDoc code and presents a simple working example.

KURT PIERSOL

OpenDoc provides a new way to write application code for the Macintosh and a number of other desktop platforms. By following the OpenDoc guidelines, you can produce applications that share files, windows, and interface elements seamlessly. The process of writing an OpenDoc application, which we call a *part handler*, is much like writing any Macintosh application. There are differences as well, of course, and this article will help you understand them.

OpenDoc applications are designed to allow code from several sources to cooperate in producing *compound documents*, documents that can embed almost any kind of content inside them. Each piece of content in the document (each *part*) includes its own part handler, the code that's used to edit and view it. To achieve this, OpenDoc part handlers must cooperate in a number of ways. They must sort out how events are passed, where data is stored on the disk, and where drawing is allowed to occur on windows or printed pages.

This article starts with a brief overview of OpenDoc and then talks about implementing a simple part handler. It will show you the absolute basics, much as TESample does for TextEdit in the Macintosh Toolbox. You'll learn about a simple example of building a part handler, included on this issue's CD: a clock that can handle two different display modes, digital and analog. The clock updates itself every second and allows the user to select the display mode from a menu.

A quick caveat: The sample code provided on the CD is from the alpha version of OpenDoc, but by the time you read this, a beta version should be available. When you begin implementing your own part handler, you may find that some details of the API have changed; however, the overall structure will be the same. The sample clock, for instance, is specific to C++ and the alpha version of OpenDoc. The final version of OpenDoc will be based on IBM's System Object Model (SOM), which will allow part handlers to be written in a variety of languages, both object-oriented and procedural. Similarly, the *XMP* prefix on OpenDoc class names (you'll see a lot of them in this article) will be changed to *OD* beginning with the beta release.

KURT PIERSOL, the chief architect of OpenDoc, previously led the Apple Event project and was an early technical lead for AppleScript. He's responsible for making technologies fit together at Apple. Kurt also likes to wear suspenders, though that has very little to do with his software architectural responsibilities. •

This is perhaps a good time to mention a bit more about SOM. This technology is the basic mechanism that OpenDoc part handlers use to communicate with one another. SOM solves many hard problems associated with using object-oriented languages, including those of subclassing across language boundaries, altering base classes under dynamic linking, and long-term maintenance of object-oriented APIs.

OVERVIEW OF OPENDOC

Before getting into the specifics of OpenDoc and how to write part handlers, we'll talk about some of the basic services you'll see in OpenDoc and where your own code fits into the OpenDoc architecture.

PART HANDLERS

Part handlers are what provide OpenDoc with its ability to handle different kinds of content in a single document. You, the developer community, will write the various part handlers that plug into OpenDoc.

Part handlers are a lot like existing applications. They handle events, draw and print, and read and store data onto disk. Every part handler provides a series of entry points that allow OpenDoc to request any of these actions from the part handler. In addition, the API has a number of "bookkeeping calls," which allow OpenDoc to provide undo services and notify part handlers when their environment has changed.

Overall, there are about 50 calls in the OpenDoc part API that a part needs to implement. This is a lot, but it actually maps fairly closely with the number of things you'd have to do to write any Macintosh application. In addition, you can ignore many of these calls in many cases. For instance, if you don't allow embedding of other parts within your part, there are about ten calls that you can safely ignore. If you don't update your display asynchronously, but simply wait for update calls, there are additional calls that you can ignore. In many typical cases, this means that you can build a part handler very quickly from existing code.

Part handlers are packaged as shared libraries in the Macintosh version of OpenDoc. This won't always be the case on other OpenDoc platforms, but you can count on the API being the same on all platforms. The alpha version of OpenDoc uses the Apple Shared Library Manager to dynamically link your part handler into OpenDoc, while the beta version will use SOM. These versions will have different linker behavior but will essentially require the same basic packaging of your code: a shared library.

In either case, you'll find that OpenDoc is an object-oriented API. That means you'll be talking to OpenDoc objects, and your part handler will itself be an OpenDoc object (or set of objects). This doesn't mean that your code has to be built from the ground up in C++, though. SOM will provide interfaces to many languages, including C.

Because part handlers are themselves objects, we often refer to them as "part objects" or "parts" in conversation. In fact, what the user would call a "part" in a document is really the combination of some persistent data stored in the document file and a set of objects that OpenDoc uses to display and manipulate the stored data. OpenDoc chooses appropriate part handlers based on the type of data stored in the document.

RUNTIME OBJECTS

As we describe how to write a part handler, we'll mention some runtime objects that interact with your code. In OpenDoc, these objects can be located at run time using the session object (XMPSession), to which your part object will be given a pointer

when it's initialized. The session object is very important because it's your link to the rest of the OpenDoc objects that are running in the document.

There's a whole list of objects that the session object makes available. Of these, only three will be important for the purposes of this article: the arbitrator, the dispatcher, and the undo stack.

- The *arbitrator* is an object of class XMPArbitrator. The arbitrator for a session is the place where part handlers register their ownership of certain resources. The menu bar, the keystroke stream, and the current selection are all examples of resources that the arbitrator tracks.
- The *dispatcher* is the object that dispatches events to the various part handlers. It's an object of class XMPDispatcher. It's used in our example as a way to register for background time.
- The *undo stack* is an object of class XMPUndo that allows OpenDoc to support multilevel undo across part handler boundaries.

Each of these objects will be discussed as it's encountered.

A RUNNING START

To give you a running start, we've built a small object-oriented framework for parts that implements the direct interface to OpenDoc. This framework is a precursor to the new part handler framework that Apple is building, and is included here simply as sample code. Our sample clock uses this framework. The good thing about the framework is that it clearly separates the work that any part handler must do to be OpenDoc compliant from the specific work performed in putting up a clock.

The framework divides the work of a part into three objects: a frame object, a facet object, and a part object. OpenDoc itself doesn't require that you create anything but a part object, but for the sake of clarity the framework divides the labor among several smaller objects. For easy reference, here's a list of the classes we'll be discussing throughout the article and their corresponding source files, included on the CD:

CPart	FWPart.h, FWPart.cpp
CFrame	FWFrame.h, FWFrame.cpp
CFacet	FWFacet.h, FWFacet.cpp
CClockPart	ClockPar.h, ClockPar.cpp
CClockFrame	ClockFra.h, ClockFra.cpp
CClockFacet	ClockFac.h, ClockFac.cpp

The classes defined by the framework generally start with the letter *C*, hence the classes CPart, CFrame, and CFacet. These three parts are helper objects for three OpenDoc classes, XMPPart, XMPFrame, and XMPFacet. XMPPart objects are OpenDoc part handlers: you'll subclass XMPPart when writing your own. OpenDoc uses the frame and facet objects to help part handlers lay themselves out in a window. How these classes work together is probably the single most complex thing to understand in OpenDoc.

XMPPart, the class from which part handlers are derived, is simply the base class of every OpenDoc part handler. It's the class that actually handles the drawing, editing, and storage. Every part handler is an implementation of some subclass of XMPPart.

CPart, in the framework, is a class derived from XMPPart. It's just a default implementation of the basic XMPPart behavior. As such, CPart is a treasure trove of information about the correct way to "ignore" calls that aren't interesting because your part handler doesn't support embedding, update asynchronously, or use offscreen bitmaps.

Every part is embedded in another part, with the exception of the *root part*, the top-level part in each compound document. When a part is embedded in another part, there's an object that's used to store information about the shape of the embedded part. This boundary between a container and an embedded part is a *frame* — an instance of the class XMPFrame. Every frame has a single part displayed inside it. The container actually embeds the frame; it knows nothing about the part inside.

Any part can be displayed in several frames at the same time. This makes it easy for a part to be visible in several windows or to have several different presentations. For example, a charting part might want to have one frame displaying the chart and another allowing the data to be edited in a table.

A *facet* (an instance of an XMPFacet object) is a visible part of a frame. There can be many facets displaying within any given frame. This is a useful property, for instance, when a container wants to "split" windows.

Both XMPFrames and XMPFacets have a field, partInfo, for storing information specific to the part being displayed. This is rather like a window refCon, a handy place to store information independent of the object itself. The CFrame and CFacet objects are designed to be plugged into the partInfo fields of their XMPFrame and XMPFacet counterparts. The containing part creates the XMPFrame and XMPFacet objects and then allows your part handler to initialize their partInfo fields. In the framework, the actual work of drawing the part on the screen is done in the CFacet object. The work of deciding what shape the embedded part will take is done in the CFrame object. As we describe the specific operations, we'll point out the class in which the code resides.

INITIALIZATION CODE

The first bit of code we'll consider is the initialization code for each part object. Each distinct part in a document gets an instance of the part object, so if there are seven little clocks running in different windows (or the same window, for that matter) there are seven instances of the clock part object. This means that you probably want to come up with a scheme to share any global data so that you aren't wasting space with many copies of it. Both the Apple Shared Library Manager and SOM support systemwide global storage, so this should be straightforward.

Resources are a special case. You'll want to be very polite about not permanently fiddling with the resource chain or making assumptions about where your resource file is in the chain. We suggest saving the previous head of the resource chain, setting your file to be the end of the chain, and using the single-level resource calls (such as Get1IndResource) to find the resources you're after. Since you'll probably want to share the resources among separate instances of your part object, it may be better to detach the resources you get and manage them yourself instead of counting on any particular application heap to have the correct resource map.

THE CONSTRUCTOR

The first step in initialization is the *constructor*. You should never do anything that could possibly fail in a constructor. This pretty much limits you to operations like

setting pointer variables to NULL, setting numeric variables to appropriate values, and making similar assignments from constants.

You can see a good example in ClockPar.cpp. The clock part simply sets up its fields with appropriate constant values.

XMPPART::INITPART

The next phase of initialization takes place in the InitPart method that every part object implements. The InitPart method is called by OpenDoc after the part object has been created, and here you *can* attempt things that can fail. This is where you should attempt to allocate any extra memory you need for your part instance, get resources if you need them, and set up your persistent storage.

Let's examine how OpenDoc's storage system looks to a part handler. When your part object is created, the InitPart method is passed a *storage unit* object in which you can persistently store information. A storage unit is really just a list of named *properties*, each of which has one or more *values*. Each value is an entire stream, like an existing Macintosh file. You can do read, write, seek, insert, and delete operations on individual values.

Each value has a type, much like the type code associated with a Macintosh file. Every property in a storage unit can have one or more values, each with their own type code. Thus, you can store multiple representations of any property. You can make up any property names you like. One special property name, kXMPPropContents, is used by OpenDoc to determine which handler goes with which part at run time. Every part object should have a property named kXMPPropContents so that OpenDoc can determine what part handler to run.

In our sample, CClockPart has an Initialize method, which is called by CPart::InitPart. It sets up the menu bar for the clock and sets up a focus set for obtaining system resources from the arbitrator (more about this later). A good example of code to set up persistent storage can be found in the implementation of CPart. The framework calls its own method, called CheckAndAddProperties, to make sure that the storage unit is set up correctly.

DRAWING CODE

Now that your initialization code is in place, you'll want to make sure you can get your part to draw onscreen. OpenDoc will call your part with the Draw method and tell you which facet should be drawn.

Our sample, CClockPart, inherits some code from CPart that asks the CFacet object to do the drawing. Notice, though, that before it does this, CPart::Draw sets up the graphics port for drawing using the clipping information from the facet. This is very similar to the basic drawing model for the Macintosh, where you draw using the appropriate graphics port and clipping region. You can find the rest of the drawing code in CClockFacet::Draw. This code consists of just the straightforward QuickDraw calls and attendant calculations needed to display either the digital or the analog clock face.

We use a utility class called CDrawInitiator to set up the drawing environment reliably. The constructor of this class does all the work of setting the graphics port's clipping region and origin. Later, the destructor restores the port to its previous state. This is a tricky bit of C++ coding that takes advantage of the object allocation behavior of stack-based objects in C++.

HANDLING LAYOUT

One of the features of CClockPart is that it presents a round shape when it's embedded. To do this, it uses the XMPFrame object's layout negotiation features. To understand this, you need to understand the notions of canvas, shape, and transform in OpenDoc.

- A *canvas* is simply a drawing context. On the Macintosh it can be either a QuickDraw graphics port or a QuickDraw GX view port.
- A *shape* is a way of describing an area of a canvas. OpenDoc supports describing shapes in terms of polygons, regions, or rectangles on the Macintosh.
- A *transform* is a geometric transformation appropriate to the type of canvas in use. In QuickDraw, the only transformation available is an offset. In QuickDraw GX, the transformations are much more powerful, capable of scaling, rotating, and offsets, as well as some more interesting transformations such as skewing.

An OpenDoc frame has a set of shapes associated with it. One, called the *frame shape*, is how the container tells an embedded object how to lay itself out. Your part handler should use the frame shape to decide what to display and how to lay it out. When your part is finished laying itself out, it can optionally specify to the container exactly what part of the frame shape it plans to use. This shape is called the *used shape* of the frame. Finally, the embedded part can specify an *active shape*, which is the subset of the frame shape that you want to use for determining whether you receive a mouse event. Often the used shape and the active shape are set to be the same shape. The container can take care of filling in any areas left untouched by the part handler.

For example, assume we have a clock part embedded in a word processor that does text wrapping. The word processor allows its embedded frames to be laid out as rectangles. When the clock is embedded, it uses the frame shape (the rectangle given by the word processor) to determine the size of the clock face. After determining the size, it sets its used and active shapes to match the shape of the round clock face. The word processor is now free to wrap text around the round clock face, and any clicks in the rectangular frame shape that aren't actually on the clock face are passed through to the word processor. Those clicks can be used to manipulate the text that's wrapped close to the clock face.

CClockPart negotiates to get the frame shape to match the clock's round face. This works but is not strictly necessary. Instead, it could simply set its used shape to match the round area of the clock. Either method will ensure that the container knows how to clip any underlying parts so that they don't draw in the clock's area.

A quick aside about shape negotiation: Negotiation is rather straightforward in OpenDoc, but knowledgeable programmers will notice that there is little support for constrained negotiation. This is not an oversight, but instead a fundamental design choice. It's up to the embedding part to constrain layout according to its model of content. This means that constraint strategies like "Boxes & Glue" or "Springs & Wires" are the province of your part handler, not OpenDoc. You can implement any of a number of layout constraint schemes on top of OpenDoc, but every part handler may constrain layout within its own frame.

You can see what happens when the container reshapes the clock in the method CClockFrame::FrameShapeChanged. CClockFrame requests a round shape from the container and then invalidates the correct areas so that redrawing occurs. For most parts, the standard behavior is to redo the layout based on the new shape, update the active and used shapes of the frame, and then invalidate the proper areas.

EVENT HANDLING

Our next area of implementation is the event handling for the clock part. This is much like writing the event handling for any Macintosh application, with one difference: you don't poll the system for events by calling WaitNextEvent. Instead, when there's an event for your part handler OpenDoc calls your part's HandleEvent method.

XMPPART::HANDLEEVENT

The code inside HandleEvent is usually a switch statement, just as in applications today. There are some minor differences, which are nicely illustrated by the code that CClockPart inherits from CPart. This code effectively delegates the various events to code that can handle each event, using a switch statement. Notice the behavior for mouse-down events, which calls CFacet::HandleMouseDown, which in turn causes the frame to become active if it isn't already.

The notion of activation in OpenDoc is closely tied to the object discussed briefly earlier, the arbitrator. An object is "active" if it owns some of the foci in the arbitrator. A *focus* is just a shared data structure or system service of some kind, such as the menu bar or keystroke stream.

When CClockFrame is told to activate, it requests a set of foci from the arbitrator. In this case, it wants the menus and selection focus. These two, with the addition of the keystroke stream, constitute the basic focus set that almost every part asks for when it wants to allow editing. You can find the code that sets up the focus set in CClockFrame::InitClockFrame. In CFrame::ActivateFrame, the focus set is requested.

Notice that the part is requesting the menu focus before it attempts to put up its menu bar. This is the basic rule to follow in all cases. If there's an arbitrator focus for the resource, you must request it and succeed in getting it before it's OK to use the resource. OpenDoc uses the arbitrator to carefully manage the sharing of data and system services, so it's very important to do the right thing and ask for foci whenever you need shared resources.

PUTTING UP A MENU BAR

One of the things that CClockPart does is to set up a menu bar. You can see the code for this in CClockPart::Initialize. The initialization code gets a reference to its menu bar object and then calls the AddMenu method of the menu bar object to add its menus. Finally, it registers command IDs to pass back when menu items are selected.

OpenDoc provides a menu bar object to help you set up menus and display them when your part has obtained the menu focus. The major reason for this object to exist at all is to support compatibility with Microsoft's proprietary OLE 2.0 document architecture. This object hides the complex menu-mixing behavior of OLE 2.0 behind a simple interface that works correctly in either an OpenDoc or an OLE 2.0 container.

Later, during execution of CFrame::FocusStateChanged, the menu bar object is asked to display itself. The actual code invoked is in CPart::InstallMenus, and basically just calls the Display method of the menu bar object.

GETTING IDLE TIME

You can get background time, delivered as idle events, on any of your frames. This is done by getting the dispatcher from the session object and registering particular frames for idle time.

You can see an example of this sort of registration in CClockPart::Initialize. In this case, the part itself is registering to receive idle events, but individual frames can also be registered. Once a frame or part has been registered for idle time, it will receive idle events in its HandleEvent method.

UNDO AND REDO

Although CClockPart is too simple to support undo, it's worthwhile to look at how you would go about adding undo support to your OpenDoc part handler. We've tried to make it as simple and unobtrusive as possible to do multilevel undo in OpenDoc.

The first step is to create code that tells OpenDoc you've done something that can be undone. You do this by getting the undo object, an instance of XMPUndo, from the session object. You then call the XMPUndo::AddActionToHistory method, which takes a hunk of data that you create to hold instructions about how to undo the latest action. OpenDoc never looks inside this hunk of bits; it merely stores it for later.

The code might look like this:

```
fSession->GetUndo()->AddActionToHistory(thisPart, myUndoData,
   kXMPSingleAction, myUndoString, myRedoString)
```

myUndoData is a pointer to the undo data, and myUndoString and myRedoString are strings to show in the Edit menu, to tell the user what action will be undone or redone.

Once the information is on the undo stack, simply calling the XMPUndo::Undo and XMPUndo::Redo methods will cause the system to send the correct messages to the parts to get the last action undone. This allows the user to undo actions that were made in other parts, without your part knowing precisely what needs to be done.

When the undo object is told to undo or redo, it calls your part handler back using the Undo or Redo method. If you never post undo actions, you never need worry about having these methods called, and you can ignore them. The XMPUndo object will always return exactly what you store in it, and it makes sure that undo and redo operations are invoked in the correct order. When the Undo object is finished with the undo data, it asks your part to dispose of it by calling your part's DisposeActionState method. This means that you can safely put pointers to other data into the undo data, since you'll get a chance to dispose of the data, and anything it points to, at a later time.

On some systems, such as one that supports persistent undo stacks, you may be asked to read and write your undo data against a persistent storage medium. This is not the case on the Macintosh, but OpenDoc does allow for it. You can safely ignore this until it becomes an issue on some platform you choose to support.

STORAGE

Eventually it becomes time to save a document. We've already discussed the OpenDoc storage environment to some degree. The storage unit object in OpenDoc is set up for the part by the OpenDoc libraries themselves, so generally a part never needs to talk directly to the file system just to read and write its own data. This system supports not only compound document storage, but also a versioning system that allows for multiple drafts.

DEALING WITH STORAGE UNITS

Once you've been given a storage unit, you typically get it ready by using the Focus call. To minimize the API, a set of common functions that can apply to the entire storage unit, a particular property, or a particular value has been abstracted out. Properties and values within a storage unit are not represented by distinct objects, but are instead captured in the focus state of the storage unit: the Focus method sets up the context for later calls. For example, the Remove method can apply to an entire property or to a single value of it, depending on whether the storage unit was focused on the property or on a value. Focusing can be absolute (when you pass a particular property ID or value index) or relative (when you pass a position code).

The read operation is performed with the XMPStorageUnit::GetValue method, and the write operation with XMPStorageUnit::SetValue. The position can be set or read with XMPStorageUnit::SetOffset and XMPStorageUnit::GetOffset. Efficient inserts and deletes can be performed with XMPStorageUnit::InsertValue and XMPStorageUnit::DeleteValue. You can also use the latter call to truncate a given value.

Typically, your part will focus on the kXMPPropContents property and do various reads or writes, depending on whether your part is being internalized (read in from storage) or externalized (written out). If your part is sufficiently large and complex, you'll probably want to use inserts and deletes to store changes to your persistent data. This has two useful effects: it makes your data more randomly accessible, and it makes the OpenDoc draft system store changes more efficiently.

This draft system allows a user to save a draft of a document and return to view the draft at any future time. Where possible, it stores only the changes between succeeding drafts, instead of storing entire copies of the document for each draft. By using OpenDoc's storage APIs, you automatically get this efficient storage of separate versions with no additional work on your part. OpenDoc only watches the storage operations, though; it doesn't attempt to detect differences on its own. If you use insert and delete operations, OpenDoc's storage system can efficiently store the changes between drafts.

BASIC I/O FOR YOUR PART

When your part is brought into memory, your InitPartFromStorage method is called, and it's passed a storage unit. You are then responsible for reading the storage unit and getting ready to receive other messages. This will happen once, and never again until the object is deleted from memory. Later, when the document is being saved, your part's Externalize method is called. You must immediately write anything you need to store persistently out into your storage unit, before returning from this method.

Your part is also free to write to its storage unit, as well as read from it, whenever it wants to. For part handlers that "virtualize" themselves from disk, this means that OpenDoc won't get in your way.

The CClockPart::InternalizeContent and CClockPart::ExternalizeContent methods are called by the framework in response to the standard methods InitPartFromStorage and Externalize. They demonstrate focusing a storage unit and doing read and write operations. CClockPart's storage needs are very simple; it just reads and writes a few flags into its storage unit.

PART INFORMATION ATTACHED TO FRAMES

As mentioned earlier, the XMPFrame objects associated with embedding have a partInfo field, which is used like a window refCon by your code. When the document

is saved, you may be asked to save the contents of this partInfo field to a particular storage unit. Your part will be called using the XMPPart::WritePartInfo method. Your responsibility is to write enough information to be able to reconstruct the partInfo field. Later, when the document is reopened, your part object will be called with the XMPPart::ReadPartInfo method. This is your cue to read the data back into memory and set up the part info for that frame object once again.

These partInfo fields are useful when you want to write a part that can have several visible frames, each with a different presentation. The chart example we used earlier is a case in point. We would want to allow a chart to be viewed as a table of data or a chart, possibly one of various chart kinds. By storing information about what to display in the frame's part info, you're freed from writing your own data structure to remember what kind of display to do in what frame. Instead, you store that information as a part of the frame's part info and implement WritePartInfo and ReadPartInfo methods to save and restore the data.

CClockPart doesn't actually use the partInfo field of its frames in a persistent fashion. It simply inherits code from CPart, which reconstructs the appropriate CFrame and CFacet objects at run time. This is completely adequate for simple parts.

BEING A GOOD OPENDOC CITIZEN

Now that we've covered the basics, there are a few last details to implement before we've got a good basic part. Since a part can have multiple frames, and a frame can be visible in multiple facets, we need to make sure our part handler does the right thing and avoids stepping on the toes of other parts.

ADDING AND REMOVING FACETS

When a part becomes visible (that is, when a facet appears), OpenDoc notifies the part with a call to the FacetAdded method. This is when your part should do any special setup it needs to (for instance, you may want to register for idle time on the frame associated with that facet). Similarly, OpenDoc calls your part handler's FacetRemoved method when the facet goes away; here you should clean up any actions you took in response to FacetAdded.

ADJUSTING MENUS

When your part handler acquires the menu focus, OpenDoc calls its AdjustMenus method. Your job is to correctly update the menus so that the right elements are checked, enabled, and so on. You can see an example in CClockPart::DoAdjustMenus, which is called by the inherited code from CPart::AdjustMenus.

RELINQUISHING ARBITRATOR FOCI

Once you've acquired any focus from the arbitrator, you'll eventually be called on to release it. This will happen via three methods: XMPPart::BeginRelinquishFocus, XMPPart::CommitRelinquishFocus, and XMPPart::AbortRelinquishFocus. The first method is called to ask your part if it's willing to relinquish a focus it owns. It should, if at all possible, say yes. It's possible, though, that you won't give up a focus, because your part object is in a mode. For instance, you wouldn't give up the serial port focus if you were in the middle of an XMODEM transfer.

Once your part has responded to the XMPPart::BeginRelinquishFocus call, you can expect another call shortly after that which informs your part that the focus has really been given to another frame, or that it hasn't. The first case is signaled by XMPPart::CommitRelinquishFocus, and the second case is signaled by XMPPart::AbortRelinquishFocus.

Occasionally, under difficult conditions, your part will simply be informed that it has either acquired a focus (through the XMPPart::FocusAcquired method) or lost a focus (through the XMPPart::FocusLost method). If your part has lost a focus, you're expected to avoid inappropriate behavior, such as attempting to adjust menus or display a menu bar when you don't have the menu bar focus.

FREEING MEMORY

Your part is expected, if possible, to free some memory on request. When it's needed, you'll be called with the XMPPart::Purge method. You're given a size that's the amount of memory requested. If you can manage it, you should free any unneeded memory from your part's data structures. Don't free anything you need to keep running, of course. You might free any resources you were holding, or free some cached data. CClockPart, our example, is so simple that it has almost nothing to purge.

IN CLOSING

By now you should have a good idea of what's involved in writing an OpenDoc part handler. As you've seen, it's much like writing an application today: you still write code to handle events, deal with storage issues, draw to the screen, and so on. The main differences are really in the "packaging" of the code and in the environment it runs in. (Some previously messy areas have even been cleanly abstracted for you. The storage system is a good example: no more ugly file handling code; you just deal with storage units and let OpenDoc handle the details.)

But the differences for users are amazing. No more worrying about which application can open which document. Instead, when they select a particular type of content to work with, the tools they need to work with that content simply appear. In user tests, many people thought that this radically wonderful technology was just a bug fix, and that it was finally working the way it was always supposed to. There can be no better indication that OpenDoc is a step in the right direction.

Thanks to our technical reviewers David Austin, Ray Chiang, Mark Minshull, Alan Spragens, and Borek Vokach-Brodsky.•

BALANCE OF POWER

Tuning PowerPC Memory Usage

DAVE EVANS

If you care about the performance of code you write for the Power Macintosh, memory usage should be your foremost concern. With the PowerPC™ 601 processor today, and even more important with future processors, memory usage of your code will have the greatest effect on its performance. Poorly written code will execute at a fraction of its potential, and often very simple changes will greatly improve the execution speed of your critical code.

Processors are improving much faster than the memory subsystems that support them. As the PowerPC chips move from 80 MHz to 100 MHz and beyond, their thirst for data to process and instructions to execute will increasingly tax memory. Memory caches attempt to mitigate that thirst, and all PowerPC processors come equipped with built-in caches. But your code can work well with a cache or it can work very poorly with a cache. I'll show you why and discuss what you can do to optimize your memory usage.

CACHE BASICS

As you know, a cache is simply very fast RAM that the processor can access quickly and that it uses to store recently referenced data and code. On the PowerPC processors, any data stored in the cache can be accessed without stalling the processor's pipeline. Accesses to data not in the cache will take about 20 times as long reading from main memory, or even 1 million times as long if the access causes a page fault with virtual memory. Getting and keeping your performance-critical code and data in the cache are therefore key to your execution speed.

A cache is divided into small blocks called *cache lines*. On the PowerPC 601, for example, the cache has 1024 cache line blocks, each holding 32 bytes. In addition, the 601 will fetch two blocks when it can, making the cache line size effectively 64 bytes.

The first PowerPC processors have set associative caches of different sizes. The 601 has an eight-way set associative, unified cache that's 32K in size, and the 603 has a two-way set associative, split cache with 8K for data and 8K for instruction code. The term *set associative* refers to the way the cache relates to main memory, which is important to your performance. In some simple caching schemes, each cache line maps directly to specific areas of main memory; any access to one of these areas loads bytes into that cache line. But on the PowerPC processor, sets of cache lines are combined and then mapped to memory. There are eight cache lines in each set on the 601, and two in each set on the 603. An access to one of the areas mapped to a set will load bytes into the last-used cache line of the set, keeping the most frequently used cache lines from being purged. This more complicated scheme typically yields much better performance than the directly mapped cache.

CACHE THRASHING

The cache will most affect your performance when you're accessing large amounts of data. A typical example of this is walking through arrays to perform some operation. The best strategy is to minimize cache collisions during your accesses, and the best tactic for this is to access your data as sequentially as possible. If you walk through memory sequentially, you'll load the cache every 64 bytes, but all 64 bytes will be available for fast processing. Here's an example:

```
unsigned long   data[64][1024];
for (row = 2; row < 64; row++)
   for (column = 0; column < 1024; column++)
      data[row][column] = data[row-1][column] +
                          data[row-2][column];
```

This example performs additions on each element of a large matrix and accesses that matrix sequentially in memory. It walks across each row, adding elements and storing the result. But just inverting the loops can significantly change the way memory is accessed:

```
unsigned long   data[64][1024];
for (column = 0; column < 1024; column++)
   for (row = 2; row < 64; row++)
      data[row][column] = data[row-1][column] +
                          data[row-2][column];
```

DAVE EVANS occasionally uses the combinatorics skills he learned at the Massachusetts Institute of Technology, but more often he's been practicing his combination punches at a Thai kickboxing gym. Designing fast algorithms for Apple's OS Platforms Group is definitely rewarding, but developing a fast left hook really gets him pumped up.•

Reversing the loops leads to less than optimal performance since we perform each addition for all the columns before moving to the next element of a row. Instead of sequential access, this access pattern jumps across memory in even steps of 4K. Unfortunately, on the PowerPC processor these accesses map to the same set of cache lines, and every operation causes the cache to reload from main memory. This second example takes twice as long to execute the same calculation on a Power Macintosh 6100/60.

By paying attention to how your code accesses memory, you can avoid serious cache thrashing like that done by the second example. Things to look out for are loops that iterate for a power of 2 steps (128, 256, and so on) and code whose memory accesses are not close together.

An approach called *blocking* may help your loops. Often your code isn't as simple as above, and your memory accesses aren't regular during the loop. If you're walking two different arrays with different increments through memory, it may be impossible to serialize your accesses. Blocking performs the calculations in blocks of rows and columns. Instead of iterating across all the columns and then proceeding to the next row, you divide the dimensional space into blocks and calculate one whole block at a time. In this next example, we calculate the multiplication of two matrices.

```
long result[64][64], foo[64][128], bar[128][64];
for (row = 0; row < 64; row++)
   for (column = 0; column < 64; column++) {
      long  sum = 0;
      for (i = 0; i < 128; i++)
         sum += foo[row][i] * bar[i][column];
      result[row][column] = sum;
   }
```

As this algorithm walks through memory, it accesses **result** and **foo** sequentially, but **bar** is accessed in 256-byte steps. Accessing **bar** by jumping through memory causes cache misses, and sequential elements of **bar** are flushed from the cache before they're needed.

By performing this operation in small blocks, we can better use the cache. The key is to use all the elements of **foo** and **bar** that are in a cache line before moving on. One way to do this is to expand the loop and perform four operations in a single iteration:

```
long result[64][64], foo[64][128], bar[128][64];
for (row = 0; row < 64; row++)
   for (column = 0; column < 64; column += 4) {
      long  sum1 = 0, sum2 = 0;
      long  sum3 = 0, sum4 = 0;
      for (i = 0; i < 128; i++) {
         sum1 += foo[row][i] * bar[i][column];
         sum2 += foo[row][i] * bar[i][column+1];
         sum3 += foo[row][i] * bar[i][column+2];
         sum4 += foo[row][i] * bar[i][column+3];
      }
      result[row][column] = sum1;
      result[row][column+1] = sum2;
      result[row][column+2] = sum3;
      result[row][column+3] = sum4;
   }
```

This expanded loop calculates a block of four cells in each iteration. This executes faster because elements of **bar** are read from the cache and don't always cause cache misses as in the earlier example. Notice that in the expanded inner loop, a cache line of the **bar** matrix will be loaded the first time that it's referenced; then the following three references to **bar** will occur without stalling. Using the **bar** elements while they're still in the cache gives us a significant improvement.

CODE STYLE

Good compilers can pay attention to your memory accesses and will optimize how you access memory. For example, load and store operations can be reordered by the compiler to occur when the data is most likely to be available. The first time data is accessed it tends to cause a cache line to load, and subsequent accesses to nearby data must also wait for the cache load to complete. The compiler may be able to help by inserting a few instructions between the loads. This way the cache line will be fully loaded when the subsequent accesses are needed.

For more information on loop expansion and instruction reordering, see the Balance of Power column in *develop* Issue 18. •

You can help your compiler by using local variables when you can. These tell the compiler exactly how the data will be used, enabling it to easily reorder the loads and stores for this data.

You should also carefully note memory dereferences, especially double dereferences. Although it may be obvious to you, the compiler often can't tell whether two pointers address the same object in memory. The compiler may be prevented from reordering instructions because it can't tell whether two operations are really dependent on each other, just because they contain dereferences. Here's an example:

```
paramBlock->size = myStructure->size;
paramBlock->offset = myStructure->offset;
```

Although it appears obvious, the compiler usually can't tell if paramBlock references the same memory as

myStructure. In the resulting binary, the compiler will be conservative and not reorder these operations for best execution. Replacing the dereference of myStructure with local variables for size and offset will allow the compiler to fully optimize this example.

INSTRUCTION THRASHING

Your code binary itself can cause the cache to thrash as it loads to be executed. This is very hard to detect and optimize. The basic problem is that your subroutines may map to the same areas of the cache, and frequent calls among them will stall to reload the cache. Some code profilers for RISC workstations have attempted to detect this problem, but for the Macintosh I can't suggest much help. Just changing the link order of your code and then executing profiles may have an effect; some link orders will thrash more than others.

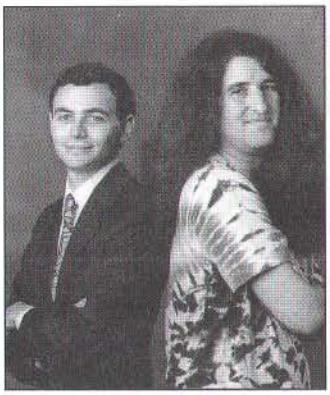
DATA STRUCTURES

The layout of your data structures can greatly affect your cache usage and your memory usage in general. For example, memory accesses that cross 64-bit memory boundaries take twice as long to process, as this forces two bus transactions. On the PowerPC 601 processor, any misaligned data access within a memory boundary takes the standard amount of time, which (because of typical Macintosh data structures) is a valuable feature of the chip; future PowerPC processors, however, may take longer to access misaligned data. If you can align your data structures, do so now. A good tactic is to keep 64-bit data at the top of your structure, followed by your 32-bit data, and so on to prevent accidental misalignment of elements. Pad the end of the structure to an even 64-bit increment if you will have arrays of structures or will allocate them on the stack. And if certain parts of your structure are accessed much more often than other parts, keep these together so that they stay in the cache, and make sure they're aligned.

DON'T FORGET

The memory usage of your speed-critical code will greatly affect its performance today, and current problems will just get worse when PowerPC processors go above 100 MHz. Profile your code to find the most critical bottlenecks; then pay close attention to how that code addresses memory. You'll be rewarded with an excellent return on your investment.

Thanks to Tom Adams, Mike Cappella, Rob Johnston, and Mike Neil for reviewing this column.•



Designing Applications for the Power Macintosh

GREG ROBBINS AND RON AVITZUR

The Power Macintosh offers surprises both for users and for developers. Users notice that it's a fully compatible Macintosh and that native applications run blazingly fast. Developers, upon learning how the Power Macintosh differs from a 680x0-based Macintosh, discover that it's still basically a Macintosh. But the Power Macintosh can offer a much richer experience than was possible with previous computers if developers break free of their old assumptions and harness the power of the machine to make software not just faster, but easier and more enjoyable to use.

The Graphing Calculator desk accessory that ships with the Power Macintosh was designed to take advantage of the machine's power to make some challenging mathematical tasks easily accessible — in particular, algebraic manipulation and 2-D and 3-D graphing. In this column we'll share some lessons we learned when writing the Graphing Calculator. We'll speak about software design rather than programming details because we frequently discovered that our intuition and the standard approaches to many aspects of the application design were no longer appropriate.

In general, we learned that it's time to break free of the idioms developed for the machines of 1984 and begin designing a new generation of software. *We're not in Kansas anymore.* The lowest common denominator of hardware — where developers usually aim in order to maintain consistency across all Macintosh models — has changed. The target has typically been an 8 MHz 68000 processor or, for color applications, perhaps a machine twice as fast. The lowest common denominator for Apple's RISC-based line of machines is a 60 MHz PowerPC 601 processor. This change represents an enormous jump: on some floating-point operations, for example, the Power Macintosh is as much as 20,000 times faster.

In our tests, calculations using the PowerPC processor's single-precision floating-point multiply-add instruction were 20,000 times faster. This means that if we had started a lengthy floating-point calculation in 1984 at the release of the Macintosh, and that calculation were still being worked on by the computer, it would take a Power Macintosh starting now just four hours to catch up. •

Even for developers targeting 680x0-based machines, a new approach to software design can dramatically improve users' experiences. The goal is to maximize use of whatever processing power is available in the design of the user interface.

Here's a summary of the tips we'd like to pass on; we'll look at each one in more detail below.

1. Tackle expensive computations when they can improve the interface.
2. Eliminate dialogs and command lines in favor of direct manipulation.
3. Drop old assumptions and idioms. Use the processing power to explore new interfaces.
4. Provide a starting point for exploration.
5. Avoid programming cleverness. Instead, assume a good compiler and write readable code.
6. Invest development time in user-centered design.
7. Learn the new rules for performance.
8. Design tiered functionality: take advantage of whatever hardware you're running on.
9. Test on real users.

THE TIPS IN DETAIL

1. Tackle expensive computations when they can improve the interface.

We took a fresh look at how to implement the visual feedback for dragging, scrolling, and zooming. Traditionally, the Macintosh has represented these with

GREG ROBBINS began writing educational software on the Apple II as a high school student. Since then, he's picked up computer engineering degrees from U.C. San Diego and the University of Pennsylvania, bagged wild boar in Fiji, and evaded sharks off Australia. When he's not consulting on Macintosh development, Greg gets way off the beaten track as a member of the Bay Area Mountain Rescue Unit. •

RON AVITZUR considered working on the Graphing Calculator to be an exercise in single-minded obsessive behavior — good training for graduate school in physics. Ron has been writing educational math software since the dawn of the Macintosh. You haven't really seen the Graphing Calculator until Ron has shown it to you; in fact, Stewart Alsop's *PC Letter* names Ron one of the first Demo Gods. •

XOR animation, which gives the impression of the action without really recalculating the window contents. This is how we first implemented them in the Graphing Calculator as well; redrawing an algebraic equation or 2-D curve, not to mention a 3-D rendered surface, is computationally expensive.

But to try to stress the processor, we tested the direct approach, and we found that the PowerPC processor could easily keep up. So now in the Calculator, when the user drags the axes, not only is the image dragged live but the exposed parts of the function are calculated and drawn as the mouse is moved. When the zoom buttons are clicked, the entire function is recomputed and redrawn several times to animate the zoom. In this way, applications can take advantage of the PowerPC processor to dramatically improve the quality of interaction in ways that are not possible on slower machines.

2. *Eliminate dialogs and command lines in favor of direct manipulation.*

Since there's processing power to burn in the PowerPC 601, we simplified the user interface by replacing many dialogs with direct manipulation. The Graphing Calculator doesn't have the dialogs typical of graphing programs for specifying the range and precision of a graph (x_{min}, x_{max}, y_{min}, y_{max}, number of points, and so on). Instead, the user controls the view of a graph through direct manipulation.

Today's computers are fast enough to allow you to implement direct interaction for complex tasks. Certain time constants play crucial roles in human factors analysis. Recognizing these thresholds can help create a smoother interface. If a task like interacting with an equation takes under one-tenth of a second, users won't be bothered by the delay; if it takes under one-fourth of a second, it's fast enough not to be annoying. Longer delays, however, make users realize that they're waiting. With fast response times, users can ignore the computer and have fun exploring the subject at hand.

By emphasizing direct manipulation, we reduced even algebraic simplification, a task that might seem to require a command-line interface, to the paradigm of MacDraw. Math is traditionally intimidating, and math programs even more so, but we wanted to make mathematics fun. This was really a user-interface challenge, and it required rethinking many fundamental assumptions. Usability was our primary design goal; functionality was second. We were pleased to discover that with direct manipulation, we could simplify the interface without giving up powerful functionality.

Since direct manipulation doesn't require users to learn any new commands or concepts, the manipulations immediately become part of a user's arsenal of tools. A powerful example of this is the drag algebra facility, which strikes many people as the most intriguing feature of the Calculator. The user can select a term of an equation and drag it elsewhere in the equation, just like dragging an object in a drawing program. The Calculator performs the algebraic manipulations necessary to keep the equation consistent. This feature immediately boosts users into a realm where they can confidently and easily manipulate an equation. Just as simple calculators did with multiplication and division, it allows users who understand the essential concepts to immediately move on to more interesting problems.

3. *Drop old assumptions and idioms. Use the processing power to explore new interfaces.*

On a Power Macintosh, you can handle hundreds or thousands of times more information than before interactively. This might allow rendered 3-D objects to become user-interface components, for example. While the Graphing Calculator flashes its buttons for the usual 8 ticks to indicate that they've been clicked, in that time it could compute and render a 3-D surface with 1000 polygons. Imagine what controls might look like if they really taxed the processing power of the machine.

Because functions are rendered so quickly on a Power Macintosh, we made 3-D surfaces spin by default. This gives users more information about the functions right away. Furthermore, there are no menus or dialogs to control the view of surfaces; just as it does for 2-D graphs, the Calculator lets users change the 3-D view by grabbing the surface with the mouse.

4. *Provide a starting point for exploration.*

Applications should avoid batch setup operations, such as requiring users to set a lot of dialog options before performing an operation. Instead, provide a starting point for exploration, with reasonable defaults for whatever's necessary to get users to that point.

Perhaps the most unusual aspect of the Graphing Calculator is what it doesn't ask of users. They don't have to set up any graphing options before viewing a curve or a surface; there are no preliminary dialogs or required commands for users to do this. For any equation that can be graphed, the user simply clicks a Graph button to draw it.

We've all had the bewildering experience of trying to use a program only to discover that we don't even know how to begin. One of the toughest problems is to

create an interface that makes functionality available and enjoyable for first-time users. But clearly this is where the design effort offers the greatest payoff.

5. *Avoid programming cleverness. Instead, assume a good compiler and write readable code.*

Cycle-counting and compiler-specific optimizations are favorite pastimes of hackers, and sometimes they're important. But we could never have completed the Graphing Calculator in under six months had we worried about optimizing each routine. Rather, we dealt with speed problems only when they were perceptible to users.

We made no attempt to look at performance bottlenecks or at the compiled code of the Calculator until after running execution profiles. We were surprised where the time was being spent. Most of the time that the Calculator is compute-bound it's either in the math libraries or in QuickDraw. So little time is spent in our code that even compiling it unoptimized didn't slow it down perceptibly. Improving our code's performance meant calling the math libraries less often.

Programmers are often tempted to spend time saving a few bytes or cycles or to fine-tune an algorithm. If the change isn't visible to users, however, the benefits may not extend beyond the programmer's satisfaction. When most of the code in an application is straightforward and readable, maintenance and improvements are easier to make. Those are changes that users *will* notice.

To maximize drawing speed without sacrificing compatibility, the Calculator renders its graphs offscreen in GWorlds and uses CopyBits to transfer them to the screen. See "Drawing in GWorlds for Speed and Versatility" in *develop* Issue 10 for a discussion of this technique. •

6. *Invest development time in user-centered design.*

Complex algorithms should be used not for their own sake but to improve the user experience. For example, as the user drags the pane divider in the Calculator window, the application redraws most of the window (offscreen in a GWorld) rather than recalculating exposed areas. The Power Macintosh is fast enough that it wasn't worth spending coding and debugging time to save on runtime calculation, because the savings wouldn't be perceived by the user.

In contrast, when users click on a 2-D curve to read an (x,y) coordinate, some quite sophisticated processing happens. As the user moves the mouse, a numeric root-finding algorithm looks for interesting points such as maxima, minima, and zero-crossings to solve equations numerically. Furthermore, because numerical methods to find maxima and minima are imprecise, we also compute symbolic derivatives of the functions and then look for where the derivative is 0, which locates maxima and minima much more accurately.

All this work goes completely unnoticed by the user. But the user does notice the result: simply clicking on the curve tells with great precision what's interesting at that point.

7. *Learn the new rules for performance.*

You may discover that there *are* places where performance tuning would be worthwhile in your application. The rules for performance have changed, and knowing the new rules is essential. Some programming techniques that traditionally improve performance can be counterproductive. In particular, on PowerPC-based systems, avoiding instruction cache misses is far more important than saving instructions.

Getting good performance out of a fast machine doesn't always come without effort. To be able to exploit modern hardware to improve an application, you must have some understanding of the hardware and what allows it to be fast. It's extremely important to understand the processor and memory architecture of your target platform.

The memory system in the Power Macintosh is much faster than the memory system of other Macintosh models. The 64-bit bus allows for substantially improved data transfer. However, the processor is much, much faster than the memory system. An uncached memory reference may take 20 times as long as a cached memory reference. Performance will actually be slowed down dramatically by a speed *optimization* that saves floating-point multiply instructions (expensive on some processors, but not on the PowerPC) at the expense of extra memory usage that forces instructions or data out of the cache.

Understanding patterns of memory reference is very important in analyzing algorithms for performance. Stepping through an array across cache lines can quickly flush all lines out of the cache. (Cache lines are discussed in the Balance of Power column in this issue.) This can cripple attempts to walk the data structures typically maintained by interface-intensive applications. The PowerPC 601 has an eight-way set associative cache, which is fairly resilient to degradation from flushing of cache lines. However, the 603 processor has just a two-way set associative cache. Any processor-intensive calculations must avoid cache thrashing if they are to avoid degrading below an acceptable level of user responsiveness.

For additional architecture insights and tuning strategies, see the Balance of Power column in *develop* Issue 18 and in this issue.•

Because no two platforms will run at the same speed, it's important to design software to work well on a variety of machines. In principle, this means that the same application could run on any Macintosh, while being far more powerful — and more pleasant to use — on a Power Macintosh.

8. Design tiered functionality: take advantage of whatever hardware you're running on.

Just as it's frustrating for users of entry-level machines to be unable to run software, it's equally frustrating for users of fast, high-end hardware with plenty of memory to have features execute unnecessarily slowly, or to be constrained by programs that expect and only take advantage of minimal resources.

The Graphing Calculator does assume a Power Macintosh as its base platform; otherwise its expectations are modest. However, its appetite is unbounded. It does all drawing using offscreen buffers in temporary memory when adequate RAM is available, but will scale back to onscreen, QuickDraw-based rendering when temporary memory is limited. System 7 makes this easy with purgeable GWorlds; once LockPixels fails, the Calculator knows that it must work within tighter constraints. Later, when it can reallocate the offscreen buffer, it does so and resumes the fast, smooth graphing effects. When memory is abundant, the Calculator uses many temporary GWorlds to buffer frames of 2-D inequality graphs calculated for varying values of n, such as the animation of $\cos nx < 0$.

The Graphing Calculator does all 2-D graph computation in 15-tick chunks. For simple curves, this typically renders the entire curve at once. But 2-D inequalities are drawn piece by piece. Once Power Macintosh computers are fast enough that an entire, complex inequality graph can be drawn in a single 15-tick time slice, users will be able to explore inequalities as fluidly as they can now play with simple functions.

To take advantage of faster machines, always base computational units on real time rather than on more arbitrary measures. If the Calculator had recomputed a fixed number of points and then called WaitNextEvent, too much time would have been yielded to other processes, even on graphs simple enough to be recomputed and drawn all at once. Instead, the Calculator calls TickCount and lets that dictate when it needs to yield time. This approach allows for a smooth user interface and cooperative execution regardless of the processor's speed.

Designing tiered functionality means abandoning assumptions about what is and is not practical on the target hardware. For addressing resources, such as available hardware and memory, or when execution can be threaded or time-sliced, the options are usually clear. For matters of raw speed, the proper approach may not be so obvious. It may come down to measuring the execution speed of a particular procedure at run time and basing a decision on that. For example, an animation effect might be suitable if it takes under 8 ticks, or a routine that bypasses QuickDraw for drawing in a GWorld may be worthwhile if it really is faster than its QuickDraw alternative.

For an example of timing graphics speed, see the article "Exploiting Graphics Speed on the Power Macintosh" in *develop* Issue 18.•

9. Test on real users.

The Graphing Calculator reflects our vision of a new kind of calculator. But user testing was essential for showing us the holes in our vision. For example, elements that were directly manipulable in our eyes were overlooked by users. So we redesigned them to make their functions clearer, and then added a demo mode to point out important controls explicitly.

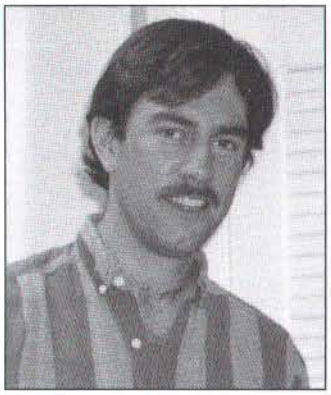
In tests, the users who looked at the help pages invariably turned first to the page at the end offering "tips." This surprised us, but also made clear where to put the most important information for using the Calculator.

Toward the end of the development of the Graphing Calculator, as the final user tests were being conducted, we saw students using the Calculator effectively within minutes, without instruction. Watching a high school student say "Wow!" as an equation came to life on the screen was probably the most satisfying moment for us as programmers. It was also the one that offered us the greatest hope about the future of personal computing. The Power Macintosh offers us the chance to reach more people while making the experience more enjoyable and easier for users than ever possible on earlier generations of computers. As developers, we have a new world to explore.

Thanks to Arnaud Gourdol, Mike Neil, Don Norman, and Alex Rosenberg for reviewing this column.•

Adding QuickDraw GX Printing to QuickDraw Applications

Now that QuickDraw GX has been released, you may be wondering what to do with your older QuickDraw applications. The good news is that by adding a relatively small percentage of code to your existing applications, you can support all the new features of the QuickDraw GX printing architecture and still retain full compatibility with non–QuickDraw GX systems.



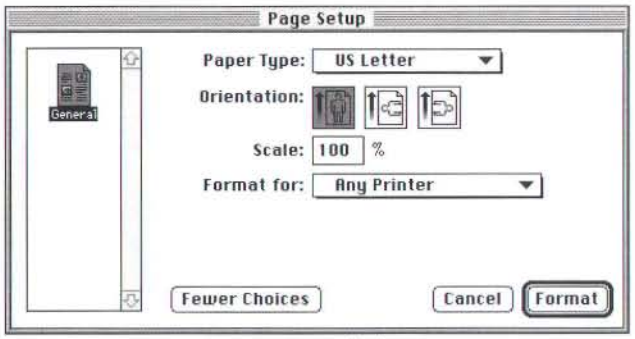
DAVE HERSEY

Compatibility with the existing application base was a primary concern during the development of QuickDraw GX; only the most hardened "print criminals" should have compatibility problems. Since a pre–QuickDraw GX application should work in both QuickDraw GX and non–QuickDraw GX environments, you may think that leaving your code as it stands is good enough. Becoming compatible probably doesn't require any revisions to your existing software, and the fact that your application will perform with or without QuickDraw GX sounds like a pretty good deal. Where's the catch?

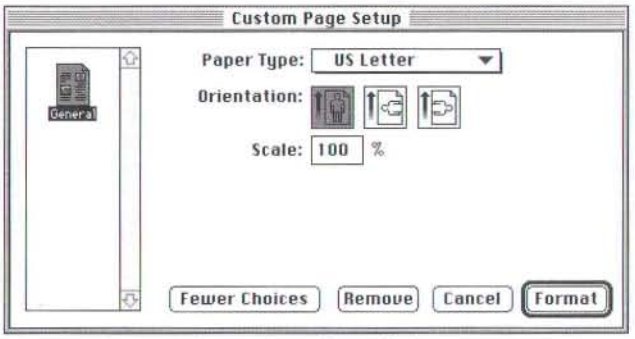
Here it is: A non–QuickDraw GX application doesn't have access to several key features of QuickDraw GX, so its users can't take advantage of many of the new printing features. For example, with QuickDraw GX a user can:

- Redirect print jobs using the new Print dialog.
- Choose page-by-page formatting, so that page 1 prints on US Letter, page 2 prints on #10 envelope, page 3 prints on landscape-oriented US Legal, and so on, instead of printing the entire document in a single format.
- Work with the new QuickDraw GX printing dialogs (shown in Figure 1) instead of the dialogs provided for compatibility with non–QuickDraw GX applications.
- Use features provided by printing extensions. Printing extensions can add items to the QuickDraw GX printing dialogs, but not to the compatibility dialogs. (Added items show up as icons in the column on the left side of each dialog.)
- Print a single copy of a document with the new Print One Copy command.
- Print documents by dragging them to desktop printers.

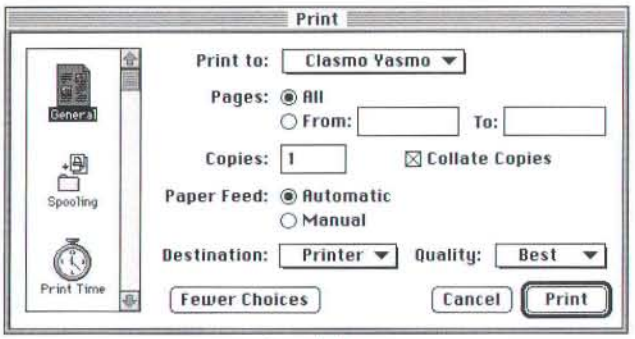
DAVE HERSEY likes the annoying buzzing noise that ImageWriters and ImageWriter LQs make. He also enjoys listening to Guns N Roses, but only while he's answering developer e-mail at Apple's Developer Support Center. Dave recently wrote a QuickDraw GX printer driver for the Apple Color Plotter (which is from an era before the first Macintosh). When asked why he bothered, he said "Well, it makes this cool wacka-wacka sound." Clearly, Dave is an audiophile for the '90s.



Page Setup dialog



Custom Page Setup dialog



Print dialog

Figure 1. The new QuickDraw GX printing dialogs

Users are shut out from all these features if they're using a non–QuickDraw GX application. In this article, you'll see how to increase an application's level of QuickDraw GX support. I'll take a QuickDraw application and convert it step by step into an application that fully supports both the QuickDraw and QuickDraw GX printing architectures. First, we'll consider some definitions and background information.

DIFFERENT LEVELS OF QUICKDRAW GX SUPPORT

To begin, let's define the different levels of QuickDraw GX support (from lowest to highest) that an application can have:

- *QuickDraw GX unaware:* The application doesn't implement any QuickDraw GX code, and QuickDraw GX translates all QuickDraw printing and drawing calls into QuickDraw GX equivalents. In other words, this is a non–QuickDraw GX application.
- *QuickDraw GX aware:* The application implements a core set of QuickDraw GX printing features but retains compatibility with documents created with earlier versions.
- *QuickDraw GX savvy:* The application implements full support for QuickDraw GX graphics, typography, and printing features. It also retains compatibility with documents created with earlier versions.
- *QuickDraw GX dependent:* The application requires the presence of QuickDraw GX graphics, typography, and printing routines to function.

Applications don't need to implement any code to be QuickDraw GX unaware. Becoming QuickDraw GX aware requires some coding, but typically not very much. Making your application QuickDraw GX aware is the *minimal* level of QuickDraw GX support that you should set your sights on — just follow the guidelines in this article.

A QuickDraw GX–aware application is really a non–QuickDraw GX application at heart, with support for QuickDraw GX printing added. This application still uses its old QuickDraw commands to represent shapes and text; however, in its print loop, the application uses the QuickDraw GX graphics translator to translate QuickDraw commands into QuickDraw GX shapes for printing, if QuickDraw GX is available.

If you're just starting out on a new project, or you're in the process of rewriting an existing product, you should consider the next level of compatibility: QuickDraw GX savvy. A QuickDraw GX–savvy application meets the requirements for being QuickDraw GX aware but also takes advantage of the extensive QuickDraw GX graphics and typography tools. You may feel that becoming QuickDraw GX savvy, although it brings with it a wealth of increased capabilities, is too big a leap to take in the short term. In that case, consider making your existing applications QuickDraw GX aware and, once they're released, going back and working on QuickDraw GX–savvy versions.

A WORD ABOUT COMPATIBILITY

It's Apple's hope and goal that no QuickDraw GX–unaware applications will be incompatible with QuickDraw GX. An existing application can be incompatible only if it relies on unsupported methods or on unpublished information that breaks under QuickDraw GX. For printing, this would include things such as trying to access the Printer Access Protocol (PAP) code from the 'PDEF' 10 resource of the LaserWriter driver. The QuickDraw GX LaserWriter driver doesn't contain a 'PDEF' 10 resource, so applications that rely on this approach must be modified to work with QuickDraw GX.

Developers were warned about the disappearance of the 'PDEF' 10 resource in "Print Hints: Looking Ahead to QuickDraw GX" in *develop* Issue 13.•

If your application has compatibility problems with QuickDraw GX, you probably already suspect it. You're a candidate for such problems if you're perpetrating any of the "printing crimes" or shaky methods that Apple has warned about in the past. Sweaty palms and pangs of guilt whenever your application is tested under new system software are likely indications that you should get out your development tools and reform your code now. (If you need the old code to run on non–QuickDraw GX systems, simply provide alternative code for running with QuickDraw GX.) Delve into the QuickDraw GX API — you're sure to find a better way of doing things.

SOME FUNDAMENTALS

Before we go into the specifics of how to add QuickDraw GX printing to your existing applications, we need to cover some fundamental ideas. I'll start with a discussion of the Collection Manager, which makes its debut with QuickDraw GX, and then briefly cover overriding messages and the Message Manager.

For a review of QuickDraw GX printing, see "Getting Started With QuickDraw GX" and "Developing QuickDraw GX Printing Extensions," both in *develop* Issue 15. See also *Inside Macintosh: QuickDraw GX Printing.*•

CREATING A FOUNDATION TO BUILD ON: THE COLLECTION MANAGER

The Collection Manager is somewhat like a memory-based version of the Resource Manager. It provides API calls that allow you to create and access *collections*, which are amorphous structures that can contain data of different types with different sizes. Collections are similar to resource files under the Resource Manager except that collections must be in memory, while resource files are by definition disk based. Routines exist for "flattening" collections into a series of bytes that can be written to disk and for "unflattening" them for later use.

Just as a resource file may contain Apple-defined data structures such as 'WIND' and 'MENU' resources, a collection may contain predefined collection items such as 'copy' and 'rang', which specify the number of copies of a print job and the page range of a print job, respectively. Like resources, collection items have attributes that are predefined (for example, the collectionLockMask attribute), but unlike resources, they also have attributes that can be programmer defined. And, as with resources, you aren't simply stuck with the core set that's been defined by Apple. Figure 2 illustrates the similarities, including a programmer-defined resource of type 'USER' and a programmer-defined collection item, also of type 'USER'.

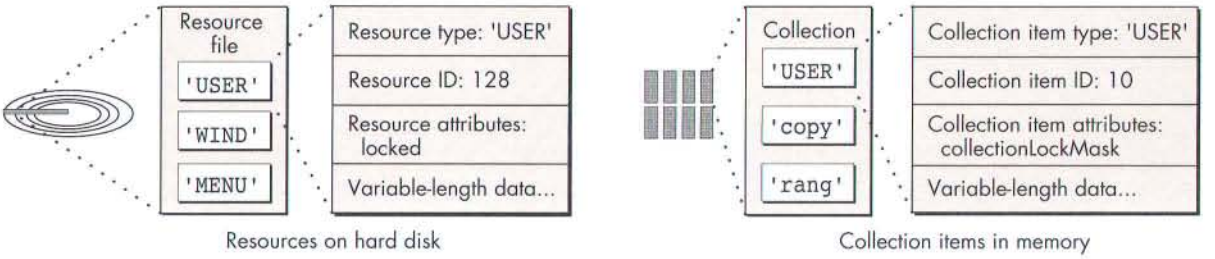


Figure 2. Similarities between resources and collection items

Items in a collection are uniquely identified in three different ways. Every item has a 4-character label, or *tag*, which identifies the type of collection item. In Figure 2, there are three collection items having the tags 'USER', 'copy', and 'rang', respectively. In addition to the tag, every collection item has a longword ID. Together, the tag and ID are one way to uniquely identify a collection item. Each collection item can be also be referenced by its collection *index* or by its tag and *tag list position*. The index is the relative position of an item in a collection, and the tag list position is the relative position of a collection item within the item's tag type — for example, the first, second, or third 'USER' item in a collection. This gives us three ways to refer to a specific collection item:

- by its tag and ID
- by its collection index
- by its tag and tag list position

It's important to note that although a collection item's index and tag list position may change as items are added or removed from a collection, an item's tag and ID will never change. Therefore, the most common way to refer to collection items is by tag and ID.

Collection tag and resource type naming conventions are the same. Apple reserves tags listed in the printing interface files, as well as those consisting entirely of lowercase letters.•

You can either create a collection with the NewCollection routine or use one of the three types of collections that QuickDraw GX automatically creates for you during printing. These types are job collections, format collections, and paper-type collections. Every time a new gxJob, gxFormat, or gxPaperType object is created, a collection is also created for that object. These collections contain items specifying the following types of information:

- job collection: number of copies, page range, destination file information (if printing to a file)
- format collection: scaling factor, page orientation, page flipping information
- paper-type collection: paper-type creator, paper-type comment

These predefined collections contain many more collection items than are shown here. See the QuickDraw GX interface file PrintingManager.h for the complete listing. By simply changing the contents of the collection items, any application, printing extension, or printer driver can easily affect how a document will be printed.

Note that all predefined collection items have the ID gxPrintingTagID = –28672.•

Your application can create and use collections for its own purposes. By no means are you restricted to using collections only within the QuickDraw GX printing architecture, although that's probably where you'll use them the most.

OVERRIDING MESSAGES FROM AN APPLICATION: THE MESSAGE MANAGER

Sam Weiss's article in *develop* Issue 15, "Developing QuickDraw GX Printing Extensions," introduced the Message Manager. The QuickDraw GX printing architecture uses the Message Manager to invoke printing operations, and your application, printing extension, or printer driver can override messages to change printing behavior. This process is explained in Sam's article, which is a good reference for anyone unfamiliar with the Message Manager and how QuickDraw GX uses it. This article assumes that you already have at least a passing familiarity with the basic concepts.

When you override a message from an application, you're more restricted in what you can do than if you override a message from a printing extension or printer driver. To understand why, consider a case where you want to override the QuickDraw GX printing message GXDespoolPage to add a serial number to the page before it's printed. You can add this override to a printing extension in a straightforward way that works regardless of which application prints the document. If, on the other hand, the override is attempted by an application, the situation becomes more problematic. The override won't be invoked because it's a despooling-phase message; only application-phase and spooling-phase messages can be overridden by applications.

To understand a second limitation of application overrides, let's look at how they're installed in the message handler chain. This is done by passing the routine GXInstallApplicationOverride a pointer to the override procedure and a reference to the document's job, like so:

```
GXInstallApplicationOverride(docJob, gxJobPrintDialog,
                                    MyJobPrintDialogOverride);
```

In this example, the application is overriding the GXJobPrintDialog message with an override function called MyJobPrintDialogOverride. docJob is the gxJob object for a

document. Because your application has access only to its own gxJob objects, an application override can be invoked only for the application that installed it. On the other hand, a message override from a printing extension can affect every document that's printed, regardless of which application printed it.

In summary:

- Application overrides are reliable only during the application and spooling phases of printing, which end once the document has been spooled to a print file. You should attempt to override only application-phase and spooling-phase messages from an application.
- Application overrides are invoked only for the application installing the override, and only for the gxJob objects in which the overrides are installed.

Application overrides are best suited for adding features to the printing dialogs and modifying the print file as a document is being spooled. If you feel limited by either of the conditions mentioned above, you should move your override out of your application and into a printing extension.

BECOMING AWARE

Now let's look at what an application developer needs to do to support QuickDraw GX printing. We'll take a QuickDraw application called Simple Sample and convert it into its QuickDraw GX–aware counterpart, Simple Sample GX. The code for both samples is on this issue's CD.

The Simple Sample application draws using various QuickDraw commands, including those for simple objects, bitmaps, and PicComments. It can handle multiple documents with multiple pages, and although it knows nothing about QuickDraw GX, it is System 7 dependent (I made it that way to save on code and confusion).

Here's a summary of what most QuickDraw GX–aware applications need to do:

- Determine whether QuickDraw GX is present, and if so, initialize the QuickDraw GX managers (and close them down when the application quits).
- Create a gxJob object for each document created and dispose of the gxJob when the document is closed.
- Override the GXPrintingEvent message (in order to support the QuickDraw GX movable modal printing dialogs) while printing dialogs are displayed.
- Update gxJob objects on resume events (in case the characteristics of the desktop printer you're printing to have changed).
- Save and load a document's gxJob object to and from disk and convert print records to gxJob objects when loading documents created from pre–QuickDraw GX versions of your application.
- Make the Custom Page Setup and Print One Copy menu items available in the File menu.
- Invoke the QuickDraw GX printing dialogs and support custom formatting by page (each page can have a unique page format, or can share another page's format).

- Store page-to-format correspondences to disk with a document (and load them again when the document is opened).
- Have a print loop that uses QuickDraw GX printing commands and translates QuickDraw commands to QuickDraw GX shapes for printing.
- Implement the Print One Copy feature.
- Support the new attribute of the 'pdoc' Apple event to properly support drag-and-drop printing of documents from the Finder.

Each of these is discussed in order in the rest of this article.

PREPARATION

Before you can do anything with QuickDraw GX, you need to determine whether it's available. You should do this in your initialize routine, after you've determined that the other managers your application requires are available. The MyInitGXIfPresent routine in Listing 1 shows one way of doing this.

Listing 1. QuickDraw GX preparation and cleanup

```
void MyInitGXIfPresent()
{
   long        gxVersion, gxPrintVersion;

   gGXIsPresent = false;
   /* Check to see whether QuickDraw GX is available. */
   if (Gestalt(gestaltGXVersion, &gxVersion) == noErr)
      if (Gestalt(gestaltGXPrintingMgrVersion, &gxPrintVersion)
                == noErr)
         gGXIsPresent = true;
   /* If so, initialize QuickDraw GX. */
   if (gGXIsPresent) {
      gClient = GXNewGraphicsClient(nil, kGraphicsHeapSize,
                (gxClientAttribute) 0);
      GXEnterGraphics();
      GXInitPrinting();
   }
}

void MyCleanUpGXIfPresent()
{
   if (gGXIsPresent) {
      GXExitPrinting();
      GXDisposeGraphicsClient(gClient);
      GXExitGraphics();
   }
}
```

In MyInitGXIfPresent, we use Gestalt to determine whether the QuickDraw GX graphics and printing routines are present. If so, we call GXNewGraphicsClient to set aside memory for QuickDraw GX graphics operations, and we call GXEnterGraphics and GXInitPrinting to enable the QuickDraw GX functions we'll

use. Note that we also set a global variable, gGXIsPresent, which indicates whether the QuickDraw GX managers we require are present. We'll check this value whenever we need to make a decision about whether to use QuickDraw or QuickDraw GX methods.

Important: You cannot intermix Printing Manager and QuickDraw GX printing calls. Do not call _PrGlue[A8FD] if you have called _InitPrinting.

When the user quits the application, we need to release any memory we allocated and close down QuickDraw GX printing and graphics. We do this as shown in the MyCleanUpGXIfPresent routine in Listing 1.

CREATING AND DISPOSING OF GXJOB OBJECTS

With the MyInitGXIfPresent and MyCleanUpGXIfPresent routines added to our code, we're now ready to make our application's documents fit into the QuickDraw GX print model. We do this by creating a gxJob object whenever we create a document. Our non–QuickDraw GX application, Simple Sample, contains a routine named MyCreateDocument that's called whenever the user creates a document. As shown in the Simple Sample GX application, this routine is modified so that when QuickDraw GX is present (as indicated by the gGXIsPresent global variable), the application creates a gxJob for each document, using the GXNewJob routine. If QuickDraw GX is not present, the application creates a print record handle (THPrint) instead.

It's common for an application to store descriptive information about a document in a private data structure, and our sample is no exception. The application uses a private structure called MyDocumentRec that contains information about the number of pages in a document, the current page being viewed, and so forth. We modify this structure also, so that we can store a document's job reference and page formatting information with the document. As seen in Listing 2, we've added the documentJob and pageFormat fields to the structure. The rest of the fields in this structure were already being managed by the QuickDraw GX–unaware application, and we'll continue to use them.

Listing 2. MyDocumentRec, modified for QuickDraw GX

```
#define kMaxPages 20        /* Max pages the sample handles. */
typedef struct MyDocumentRec {
      THPrint     documentPrintHdl; /* Print record for document. */
      gxJob       documentJob;      /* Job for document. */
      gxFormat    pageFormat[kMaxPages]; /* Format for each page. */
                                    /* If nil, we use the job format. */
      long        numPages;         /* Number of pages in document. */
      long        curPage;          /* The current page displayed. */
      FSSpec      documentFSSpec;   /* Document's file spec. */
      Str31       documentTitle;    /* The title of this document. */
      WindowPtr   documentWindow;   /* The window for this document.*/
} MyDocumentRec, *MyDocumentPtr;
```

We also need to modify the application's MyDisposeDocument routine, which is called whenever a document is closed. In the modified routine, we dispose of the document's gxJob.

The changes that we've made so far are indicative of the approach we'll continue to use as we make our sample QuickDraw GX aware: adding code that executes only if QuickDraw GX is present. If the user isn't running QuickDraw GX, our converted sample will appear and function as the original did. But if QuickDraw GX is present, all the new functionality will kick in.

OVERRIDING GXPRINTINGEVENT

When we added support for gxJobs in the MyCreateDocument routine, we also added the following line of code:

```
GXInstallApplicationOverride((*createdDocument)->documentJob,
                             gxPrintingEvent, MyPrintingEventOverride);
```

Every application that aspires to be QuickDraw GX aware should include such a line. Remember, QuickDraw GX has movable modal printing dialogs; the GXPrintingEvent message is sent whenever a dialog is moved. Unless you override GXPrintingEvent, you won't have a chance to update your application's windows when the dialogs are moved.

The code to support window updates when the printing dialogs are moved is actually quite simple. You just need to add a routine that overrides the GXPrintingEvent message and calls your event-handling routine (Listing 3), and to make sure that you've disabled the appropriate menu items before displaying the new printing dialogs. Note that your override should not forward the GXPrintingEvent message, but instead should perform a total override of it.

Listing 3. MyPrintingEventOverride

```
OSErr MyPrintingEventOverride(EventRecord *anEvent, Boolean filterEvent)
{
   /* Handle events in whatever way is appropriate. MyDoEvent is
      our generic event handler. We don't pass it events that it
      shouldn't handle while printing dialogs are displayed. */
   if (!filterEvent)
      switch (anEvent->what) {
         case mouseDown:
         case keyDown:
         case autoKey:
            break;
         default:
            MyDoEvent(anEvent);
      }
   return noErr;
}
```

UPDATING GXJOB OBJECTS ON RESUME EVENTS

There's another piece of event-handling code that every QuickDraw GX–aware application should include. To support the reentrant nature of QuickDraw GX, you must call GXUpdateJob on each gxJob that your application is using whenever you receive a resume event. (See the code for Simple Sample GX on the CD.) This enables QuickDraw GX to update the information in the gxJob in case it changed while your application was in the background. As an example, suppose that a user suspends your application to change the printing extension setup for the destination

desktop printer. Upon switching back to your application, this user will expect any open documents to use these new settings at print time. Unless you call GXUpdateJob on your gxJob objects, the new settings won't be there.

SAVING AND LOADING A DOCUMENT'S GXJOB

Now that we can create gxJob objects for new documents, let's take a look at how we can save these to disk and later retrieve them when the documents are opened. We want to save them for the same reason that we want to store print records with documents under non–QuickDraw GX systems: if a user has gone to the trouble of configuring print settings for a document, the user-friendly thing to do is to use those settings the next time the document is opened.

Because the data we want to save is stored in an abstract data structure (gxJob), we need a way to convert it to a more tangible form. We accomplish this with the GXFlattenJob or GXFlattenJobToHdl routine. GXFlattenJob passes the converted data as a stream of bytes, whereas GXFlattenJobToHdl places the flattened data in a handle. You would typically use GXFlattenJob to store the flattened gxJob in a data fork, but GXFlattenJobToHdl to save it as a resource.

Since our application stores its print records in resources, we'll store its flattened gxJob objects in resources as well. To do this, we modify our application's MySavePrintInfo routine, which is called to save print records to disk. If QuickDraw GX is present, the routine will instead save our flattened gxJob.

In the following code, we flatten a gxJob into a handle called thePrintData, which can then be written to the disk as a resource.

```
thePrintData = NewHandle(0);
GXFlattenJobToHdl(whichDocument->documentJob, thePrintData);
```

Next, we modify the application's MyLoadPrintInfo routine, which is used to retrieve print records that were previously saved with MySavePrintInfo. This routine must do several things, based on whether QuickDraw GX is present and whether a gxJob or print record has been previously stored with a document. The flow of control is shown in Listing 4.

Listing 4. MyLoadPrintInfo flow of control

```
if (gGXIsPresent) {
   if there's a previously saved gxJob
      unflatten it and use it
   else
      if there's a previously saved print record
         convert that to a gxJob and use it
      else
         use the default gxJob we created in MyCreateDocument
}
else {
   if there's a previously saved print record
      use it
   else
      use the default print record we created in MyCreateDocument
}
```

As it turns out, some of the steps in Listing 4 can be combined. For example, regardless of whether QuickDraw GX is present, we may need to retrieve a previously saved print record. What?! Accessing old-style print records when QuickDraw GX is present? Sounds strange, doesn't it? We need to do this if QuickDraw GX is available and the user opens a document that contains a print record but not a gxJob (because it was created with an older version of the application). We convert the print record to a gxJob with the GXConvertPrintRecord routine, so that the gxJob has as many of the old print record's settings as possible.

CONFIGURING AND HANDLING MENUS

We need to alter our menu routines so that the Custom Page Setup and Print One Copy commands are available to QuickDraw GX users. We might decide to have two different File menus, the installation of which would depend on whether QuickDraw GX is present. Or we might have only one File menu and add or subtract items from it when it's installed in the menu bar. The approach that's best for a particular application depends on how the application is structured. (The same choice of approaches applies to other menus that might need to be modified based on whether QuickDraw GX is available.)

In the sample code, we modify the File menu as follows: the application contains a 'MENU' resource for the QuickDraw GX version of the File menu; if gGXIsPresent is false, we remove the Print One Copy and Custom Page Setup menu items from this menu.

```
fileMenu = GetMHandle(mFile);
DelMenuItem(fileMenu, iPrintOneCopy);
DelMenuItem(fileMenu, iCustomPageSetup);
```

The order of the deletions is very important! If we deleted in the reverse order, the Custom Page Setup menu item would be deleted, but when we tried to delete Print One Copy, we would actually delete whatever came *after* Print One Copy. Why? Because the menu would be one item shorter, and the index into it would no longer be valid. Just delete menu items from bottom to top and you'll be fine.

Monkeying around with the placement (and inclusion) of menu items like this will throw our menu-enabling and menu-selection routines out of whack. We need to make sure that regardless of whether the Print command is item number 8 or 9 in the menu, we still treat it as a Print command. Again, the approach to use will depend on the application.

It's conceivable that you would use macros and would make the same changes to both the menu-enabling and menu-selection routines, but in the sample I chose to tackle menu enabling and disabling differently than I did menu handling. For the simple enabling and disabling of menus in the MyAdjustMenus routine, I check to see whether QuickDraw GX is present and adjust the item numbers based on that. This is the easiest approach because it requires only minor changes to the routine.

The menu selection situation is a little different. Typically, applications contain a routine that uses a switch statement based on the ID of the menu and menu items chosen. This means that we'd need either two such routines (one for QuickDraw and one for QuickDraw GX) or a different approach from what we used in MyAdjustMenus. I opted for a different approach.

The sample application's MyDoMenuCommand routine uses a switch statement (as most applications do) based on the menu ID and menu item extracted from the result of the MenuSelect and MenuKey routines. When QuickDraw GX is not present, the

Quit menu item will be item 10 in the File menu; otherwise, it will be at item 12 (see Figure 3). The switch statement in the MyDoMenuCommand function compares the menu item selected to the items in the QuickDraw GX version of our File menu. Therefore, it will expect that item 12 will be Quit. Item 10 will be interpreted as Print One Copy! This would be disastrous — the application would not quit, and our QuickDraw GX–specific code would be executed when QuickDraw GX wasn't around! That's not likely to be a pleasant user experience.

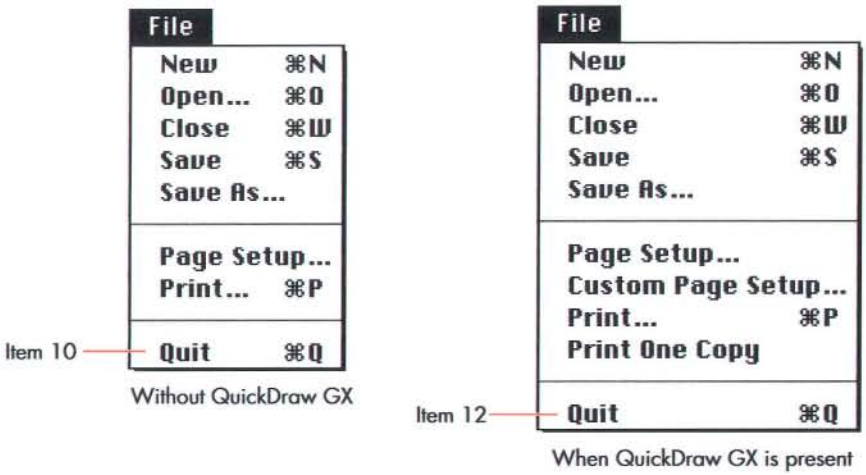


Figure 3. The File menu without QuickDraw GX and with QuickDraw GX

A new routine, MyConvertMenuItem, solves this problem (Listing 5). This routine is called just before we enter the switch statement in MyDoMenuCommand. If QuickDraw GX is present, MyConvertMenuItem does nothing; otherwise, it checks to see if the menu item selected was affected by our deletion of the QuickDraw GX menu items, and if so adjusts it. How's that for an easy solution?

Listing 5. MyConvertMenuItem

```
void MyConvertMenuItem(short *menuID, short *menuItem)
{
   if (!gGXIsPresent) {
      if (*menuItem == iCustomPageSetup)
         *menuItem = iPrint;          /* Print was selected. */
      else
         if (*menuItem == iPrintOneCopy)
            *menuItem = iQuit;        /* Quit was selected. */
   }
}
```

INVOKING THE PRINTING DIALOGS AND SUPPORTING CUSTOM FORMATTING

Earlier, we added code to our application to support window updates when the printing dialogs are displayed. Now, let's discuss the code required to actually display the dialogs.

There are now three printing dialogs instead of two (as shown earlier in Figure 1). The Page Setup dialog now allows users to modify a document's default page format.

The new dialog, Custom Page Setup, provides a way to change page formatting on a page-by-page basis. If your application creates documents that can have only a single page, implementing the Custom Page Setup dialog isn't necessary; the Page Setup dialog can be used to configure the document's only page format. The Print dialog is similar to its non–QuickDraw GX counterpart, although much enhanced.

We use GXJobDefaultFormatDialog, GXFormatDialog, and GXJobPrintDialog to display the Page Setup, Custom Page Setup, and Print dialogs, respectively.

We modify our MyDoPageSetup routine to call GXJobDefaultFormatDialog instead of PrStlDialog when QuickDraw GX is present. In our MyDoCustomPageSetup routine, which is called only when QuickDraw GX is available, we simply call GXFormatDialog, passing the current page's gxFormat.

In both of these page setup routines, and in the code for MyPrintDocument that follows, we call a function named MyAdjustMenusForPrintDialogs, which disables and enables entire menus (Listing 6). We disable menus before displaying a printing dialog, and enable them once the dialog goes away. If we didn't disable the menus, users would be able to select menu items. And, because the GXPrintingEvent override calls our MyDoEvent routine, any queued-up menu selections would be processed when the GXPrintingEvent message was sent. The user could be in the Print dialog, then select Quit from the File menu, and the next time the window was updated, the application would quit! The only menus a user should have access to are the Edit menu and the system menus. (Users can open desk accessories from the Apple menu, for example, while a printing dialog is displayed — but they should not be able to open the About item for the application.)

Listing 6. MyAdjustMenusForPrintDialogs

```
void MyAdjustMenusForPrintDialogs(Boolean dialogGoingUp)
{
   MenuHandle  appleMenu, fileMenu, editMenu, documentMenu;

   appleMenu = GetMHandle(mApple);
   fileMenu = GetMHandle(mFile);
   editMenu = GetMHandle(mEdit);
   documentMenu = GetMHandle(mDocument);
   if (dialogGoingUp) {
      DisableItem(appleMenu, iAbout);
      DisableItem(fileMenu, 0);
      DisableItem(documentMenu, 0);
      HiliteMenu(0);
   }
   else {
      EnableItem(appleMenu, iAbout);
      EnableItem(fileMenu, 0);
      DisableItem(editMenu, 0);
      EnableItem(documentMenu, 0);
   }
   DrawMenuBar();
   gInPrintDialog = dialogGoingUp;
}
```

The system enables some menus, namely the Help menu, the Application menu, and the Keyboard menu, even when dialogs are displayed. Your code doesn't need to deal with them.•

The MyPrintDocument routine (Listing 7) displays the appropriate Print dialog and then branches to our QuickDraw or QuickDraw GX printing routine. Note that we've been careful to call PrOpen and PrClose only when QuickDraw GX is not present. As mentioned earlier, you cannot intermix Printing Manager and QuickDraw GX printing calls.

Listing 7. MyPrintDocument

```
OSErr MyPrintDocument(MyDocumentPtr whichDocument)
{
   OSErr            err = noErr;
   gxEditMenuRecord editMenuRec;
   gxDialogResult   result;

   if (gGXIsPresent) {
      /* If GX is present, fill in the location of the application's
         Edit menu items, enable/disable the appropriate menu items,
         and display the Print dialog. If the user clicks OK, print. */
      editMenuRec.editMenuID = mEdit;
      editMenuRec.cutItem    = iCut;
      editMenuRec.copyItem   = iCopy;
      editMenuRec.pasteItem  = iPaste;
      editMenuRec.clearItem  = iClear;
      editMenuRec.undoItem   = iUndo;
      MyAdjustMenusForPrintDialogs(true);
      result = GXJobPrintDialog(whichDocument->documentJob,
                  &editMenuRec);
      MyAdjustMenusForPrintDialogs(false);
      if (result == gxOKSelected)
         err = MyGXPrintLoop(whichDocument);
   }
   else {
      /* If GX is NOT present, open the printer driver and print
         using the Printing Manager. */
      PrOpen();
      if (PrJobDialog(whichDocument->documentPrintHdl))
         err = MyQDPrintLoop(whichDocument);
      PrClose();
   }
   return err;
}
```

We've made a few other changes behind the scenes; see the complete code on the CD for details. The code for repaginating a document has been changed slightly to use the dimensions of QuickDraw GX page formats, if available, rather than those in the rPage field of the old print record. We also made minor changes to our MyInsertPage and MyDisposePage routines in order to manage gxFormats on a page-by-page basis. In the Simple Sample GX application, we store nil in our MyDocumentRec.pageFormat array whenever a new page is created. Because nil is

an invalid gxFormat reference, we can use it to tell the application to use our gxJob object's default format for a given page. If the value stored is non-nil, we can safely assume that it's a valid reference to a custom page format. Finally, in our MyDisposePage routine, we now call GXDisposeFormat if the page being removed uses a custom page format.

Since we store gxFormat references on a page-by-page basis, shouldn't we also dispose of page formats in the MyDisposeDocument routine? Well, we certainly can do that, but there's really no need. When we dispose of the document's gxJob, QuickDraw GX automatically disposes of all of the job's page formats. The flip side of this is that you must not attempt to use any gxFormat references once the format's corresponding job is disposed of.

Continuing with this line of thought, what happens to a document's page formats when the document is saved to disk and later reopened? As it turns out, flattening a gxJob causes all of the job's page formats to be flattened also. This means that the code we wrote earlier to flatten and store a document's gxJob in a resource will also store page formats. When we later unflatten the job, the page formats will be unflattened as well.

STORING THE PAGE-TO-FORMAT CORRESPONDENCES

It's easy to fall into the trap of thinking that you don't need to do anything more to support saving and loading of by-page formats — but don't. There are still two more issues to consider.

The first issue is straightforward enough: we haven't stored our page-to-format correspondences, so the next time we open the document we'll have no idea which format goes with which page. The second issue requires a bit more explaining.

When QuickDraw GX creates a page format, it returns a format reference that's based partially on the reference of the job with which the format is associated. Since reference IDs for gxJob objects differ depending on conditions when the job is created, a job reference that's valid when a document is saved is unlikely to be correct when the document is later reopened. Similarly, the format references we have when a document is saved are unlikely to be correct when the document's job is later unflattened.

It should now be clear that we can't simply store our page format reference IDs with a document. To provide the page-to-format information we'll need when we open the document, we have to find another method. Fortunately, there's a very easy way to do this via the Collection Manager.

As mentioned earlier, every gxFormat has a collection associated with it. QuickDraw GX creates these collections automatically when a gxFormat is created. When a format is flattened, its collection is also flattened. Specifically, any collection items that have the collectionPersistenceBit attribute set are included in the flattened data stream. This attribute is set by default when a new collection item is added, so you need to change it only if you *don't* want a collection item to be included in the flattened data.

To store page-to-format mapping, we'll create a custom collection item. We'll store this collection item in the default format's format collection. The collection item consists of an array of long words — one for each page in the document. In this array, we store the index of the format to use for each page, in order. Since format *indices* are preserved during flattening (unlike format *references*), we'll be able to reconstruct the page-to-format relationships when we reopen the document. The

MySaveFormatRefs routine (Listing 8) saves the format indices. This routine is called from our MySavePrintInfo routine, just before we flatten the document's gxJob (and in turn its page formats and format collections).

Listing 8. MySaveFormatRefs

```
#define kMyFormatInfoType     'FLST'
#define kMyFormatInfoTagID    1000
OSErr MySaveFormatRefs(MyDocumentPtr whichDocument)
{
   OSErr          err = noErr;
   Handle         theFormatIdxList;
   Collection     fmtCollection;
   gxFormat       defaultFmt;

   if (whichDocument->numPages > 0) {
      /* Get the job's default format's collection. */
      defaultFmt = GXGetJobFormat(whichDocument->documentJob, 1);
      fmtCollection = GXGetFormatCollection(defaultFmt);
      /* Create a list of page-to-format correspondences for the
         current document. If there are no errors, add the item to
         the job's default format's collection for later retrieval. */
      err = MyCreateFormatIndexList(whichDocument, &theFormatIdxList);
      if (err == noErr) {
         HLock(theFormatIdxList);
         err = AddCollectionItem(fmtCollection, kMyFormatInfoType,
                  kMyFormatInfoTagID, GetHandleSize(theFormatIdxList),
                  *theFormatIdxList);
         DisposeHandle(theFormatIdxList);
      }
   }
   return err;
}
```

Our saved documents now contain information for associating pages with page formats. The MyAdjustFormats routine (Listing 9) extracts this information from the default format's format collection when we load the saved document. In effect, the code finds new format reference IDs for each format we flattened and stores those IDs with the pages that use them. In this way, we completely avoid relying on the old (and invalid) format references.

TRANSLATING AND PRINTING QUICKDRAW COMMANDS

At long last, we're ready to look at the code that translates our QuickDraw commands to QuickDraw GX shapes and prints them.

To do the translation, we'll use the QuickDraw GX translator routines. We specify how we would like the translation to be performed by passing one of the gxTranslationOptions to the translator routines. (The routines and translation options are listed in *Inside Macintosh: QuickDraw GX Environment and Utilities.*) Normally, the default translation options are all you need, and those are what we use in Simple Sample GX.

Listing 9. MyAdjustFormats

```
OSErr MyAdjustFormats(MyDocumentPtr whichDocument)
{
   OSErr       err = noErr;
   Handle      theFormatIdxList = nil;
   gxFormat    theFormat, defaultFmt;
   long        pg, numPages, fmtIdx, *idxList, idx, listSize, attribs;
   Collection  fmtCollection;

   defaultFmt = GXGetJobFormat(whichDocument->documentJob, 1);
   fmtCollection = GXGetFormatCollection(defaultFmt);
   err = GetCollectionItemInfo(fmtCollection, kMyFormatInfoType,
                  kMyFormatInfoTagID, &idx, &listSize, &attribs);
   if (err == noErr)
      theFormatIdxList = NewHandle(listSize);
   if (theFormatIdxList != nil) {
      HLock(theFormatIdxList);
      err = GetCollectionItem(fmtCollection, kMyFormatInfoType,
               kMyFormatInfoTagID, dontWantSize, *theFormatIdxList);
      numPages = listSize / sizeof(long);
      idxList = (long *) *theFormatIdxList;
      for (pg = 0; (err == noErr) && (pg < numPages); pg++) {
         fmtIdx = idxList[pg];
         if (fmtIdx != (long) nil) {
            theFormat = GXGetJobFormat(whichDocument->documentJob,
                                       fmtIdx);
            err = GXGetJobError(whichDocument->documentJob);
         }
         else
            theFormat = nil;
         if (err == noErr)
            whichDocument->pageFormat[pg] = theFormat;
      }
      DisposeHandle(theFormatIdxList);
   }
   return err;
}
```

The translation routines come in two varieties — those that take a single QuickDraw PicHandle and convert it to a QuickDraw GX picture shape, and those that let you execute QuickDraw commands and create equivalent QuickDraw GX shapes as you do so. Converting a PicHandle to a gxPicture shape is straightforward; therefore, we're going to take the second approach. Most applications don't print by simply making a QuickDraw DrawPicture call, so understanding how to convert individual QuickDraw commands to QuickDraw GX shapes "on the fly" is probably more useful. If your needs are different, you can use the GXConvertPICTToShape routine to convert a PicHandle into a gxPicture shape.

The routines we'll use are GXInstallQDTranslator and GXRemoveQDTranslator. GXInstallQDTranslator tells QuickDraw GX to begin translating QuickDraw commands into QuickDraw GX shapes, and GXRemoveQDTranslator tells QuickDraw GX that we've completed drawing. These routines are used in conjunction with a third routine, called a gxSpoolProc, which you create. Your

gxSpoolProc routine will have the following format and will be called whenever QuickDraw GX completes a new shape during the translation:

```
OSErr MyShapeSpoolProc(gxShape currentShape, long refCon);
```

The currentShape parameter contains the QuickDraw GX shape that the translator just created, and the refCon parameter is a programmer-defined value that you pass to GXInstallQDTranslator. You needn't use the refCon parameter (you can pass nil), but as we'll see in a moment, the refCon can be very handy.

The code in Listing 10 shows the basic QuickDraw GX print loop, with support added to translate QuickDraw commands into QuickDraw GX shapes.

The code in Listing 10 contains nrequire, require, nrequire_action, and require_action macros, which are discussed in the article "Living in an Exceptional World" in *develop* Issue 11. These macros, which don't require QuickDraw GX themselves, are now included in the QuickDraw GX interface file GXExceptions.h. •

Listing 10. MyGXPrintLoop

```
OSErr MyGXPrintLoop(MyDocumentPtr whichDocument)
{
   OSErr            err;
   long             firstPage, lastPage, numPages, pg;
   short            oldPage;
   gxViewPort       printViewPort;
   Point            patStretch = {1,1};
   gxFormat         pageFormat;
   Rect             everywhereRect;
   gxShape          pageShape;
   MySpoolDataRec   spoolData;

   oldPage = whichDocument->curPage;
   /* Determine which pages the user selected to print, and print
      only those pages that are actually in the document. */
   GXGetJobPageRange(whichDocument->documentJob, &firstPage, &lastPage);
   if (lastPage > whichDocument->numPages)
      lastPage = whichDocument->numPages;
   /* Calculate the number of pages to print and begin printing. */
   numPages = lastPage - firstPage + 1;
   err = GXGetJobError(whichDocument->documentJob);
   nrequire(err, PageRangeError);
   GXStartJob(whichDocument->documentJob, whichDocument->documentTitle,
              numPages);
   err = GXGetJobError(whichDocument->documentJob);
   nrequire(err, StartJobFailed);
   /* Create a new view port for printing and set our translator
      rects to "wide open" so that they include all data we're
      drawing. For each page we print, call GXStartPage, draw,
      and call GXFinishPage. */
   SetRect(&everywhereRect, 0, 0, 32767, 32767);
   printViewPort = GXNewViewPort(gxScreenViewDevices);
```

(continued on next page)

Listing 10. MyGXPrintLoop *(continued)*

```
   for (pg = firstPage; (err == noErr) && (pg <= lastPage); pg++)
   {
      /* Get the page's format and start printing the page. */
      pageFormat = whichDocument->pageFormat[pg - 1];
      if (pageFormat == nil)
         pageFormat = GXGetJobFormat(whichDocument->documentJob, 1);
      GXStartPage(whichDocument->documentJob, pg, pageFormat, 1,
            &printViewPort);
      err = GXGetJobError(whichDocument->documentJob);
      /* If there were no errors, set up the translator, draw
         the QuickDraw data for current page, and remove the
         translator. */
      nrequire(err, StartPageFailed);
      spoolData.printViewPort = printViewPort;
      GXGetFormatDimensions(pageFormat, &spoolData.pageArea, nil);
      GXInstallQDTranslator(whichDocument->documentWindow,
            gxDefaultOptionsTranslation, &everywhereRect,
            &everywhereRect, patStretch, MyPrintAShape, &spoolData);
      whichDocument->curPage = pg;
      SetPort(whichDocument->documentWindow);
      MyDrawContents(whichDocument->documentWindow);
      GXRemoveQDTranslator(whichDocument->documentWindow, nil);
      GXFinishPage(whichDocument->documentJob);
   }
StartPageFailed:
   GXFinishJob(whichDocument->documentJob);
   err = GXGetJobError(whichDocument->documentJob);
   GXDisposeViewPort(printViewPort);
   whichDocument->curPage = oldPage;
StartJobFailed:
PageRangeError:
   return err;
}
```

Several important things are going on in the code in Listing 10. You may recognize the basic QuickDraw GX print loop, which consists of everything in MyGXPrintLoop except the view port and translator routines. The first thing we do in the print loop is create a gxViewPort object, because GXStartPage needs to know which view ports we'll be drawing to. Only shapes drawn in the specified view ports will be printed. For our purposes, one view port will suffice, so that's all we create.

There are two basic print loop methods for QuickDraw GX: the first uses GXPrintPage to print a single gxPicture shape; the second method uses the GXStartPage and GXFinishPage routines. In the second method, the application specifies a list of view ports, and any QuickDraw GX drawing that occurs in any of these view ports is instead redirected to a print file. Since our application draws several shapes on a page, it makes sense to use the GXStartPage and GXFinishPage approach. If we had only one shape to print (for instance, if we had used GXConvertPICTToShape), or if our gxSpoolProc collected all the converted shapes into one gxPicture shape, using GXPrintPage would make more sense.

Notice that the custom page formats that we've added to our application are supported, and they require only a couple of lines of code. Recall that in this application we've decided to use nil to represent the default job format. If the current page's format reference ID is not nil, we pass GXStartPage the reference; otherwise, we pass the gxJob object's default format. This default format is always positioned as the first format in a gxJob, so we can obtain it as follows:

```
pageFormat = GXGetJobFormat(whichDocument->documentJob, 1);
```

Before we can issue our QuickDraw drawing commands, we must call GXInstallQDTranslator. Because all QuickDraw drawing calls are ignored by the QuickDraw GX printing routines, we need to translate all QuickDraw commands to QuickDraw GX shapes for printing. In the GXInstallQDTranslator call, we specify that we want to use the default translation options, that we don't want to stretch patterns, and that our shape-handling routine is called MyPrintAShape. Finally, remember the refCon parameter we discussed earlier? Well, here's where it comes into play. In the refCon, we pass a pointer to a MySpoolDataRec, which is defined as

```
typedef struct MySpoolDataRec {
      gxRectangle    pageArea;         /* Page rectangle. */
      gxViewPort     printViewPort;    /* View port we're printing in. */
} MySpoolDataRec, *MySpoolDataPtr;
```

The MyPrintAShape routine is passed each QuickDraw GX shape that is created as the result of the QuickDraw translation. We can print each shape because a pointer to our MySpoolDataRec is passed in the refCon parameter of MyPrintAShape. We print a shape by attaching the MySpoolDataRec.printViewPort to the current shape, and then drawing the shape. We use the page rectangle in the MySpoolDataRec to determine whether a translated shape will appear on the printed page. If the shape isn't on the page, it doesn't make sense to waste time and disk space spooling it. Listing 11 shows how cleanly this all fits together.

Listing 11. MyPrintAShape

```
OSErr MyPrintAShape(gxShape currentShape, long refCon)
{
   MySpoolDataPtr    spoolData;
   gxShapeType       theShapeType;

   spoolData = (MySpoolDataPtr) refCon;
   theShapeType = GXGetShapeType(currentShape);

   /* Don't waste time spooling the shape if it's being drawn off
      the page. */
   if ((theShapeType == gxEmptyType) || (theShapeType == gxFullType) ||
         (theShapeType == gxPictureType) ||
         GXTouchesBoundsShape(&spoolData->pageArea, currentShape)) {
      GXSetShapeViewPorts(currentShape, 1, &spoolData->printViewPort);
      GXDrawShape(currentShape);
   }
   return (OSErr) GXGetGraphicsError(nil);
}
```

Back in our print loop, we simply draw our page's QuickDraw representation between the GXInstallQDTranslator and GXRemoveQDTranslator calls. QuickDraw commands are translated to QuickDraw GX shapes and printed in one fell swoop.

PRINT ONE COPY

A soon-to-be-familiar sign of the QuickDraw GX application will be the Print One Copy command in the application's File menu. The option to print one copy of a document without any dialogs is a big convenience to the user, and it requires only a few minutes of coding to support.

In addition to changing the necessary menu setup and handling routines, we need to add a routine to support Print One Copy (Listing 12). In this routine, we temporarily reset three of the printing options that the user may have previously changed. First, we set up the print job so that it prints only one copy. The number of copies last printed is stored in the gxJob object, and we want to make sure that if the user previously printed multiple copies of a document, only one copy comes out of the printer when Print One Copy is chosen. Second, we indicate that we want to print all pages of the document, rather than the last page range used. Finally, the output should *come out of the printer*. If the job was last printed to a file, we'll need to change the job object's "Print to disk" setting. Once again, we call upon the Collection Manager, although this time we access the job collection.

Listing 12. MyPrintOneCopy

```
OSErr MyPrintOneCopy(MyDocumentPtr whichDocument)
{
   OSErr                  err;
   Collection             jobCollection;
   gxCopiesInfo           copiesInfo;
   gxFileDestinationInfo  destInfo;
   gxPageRangeInfo        pageRangeInfo;
   Ptr                    oldCopiesInfo = nil, oldPageRangeInfo = nil,
                          oldDestInfo = nil;
   long                   oldCopiesSize, oldPageRangeInfoSize,
                          oldDestInfoSize;

   /* Get the job collection and set it up to print one copy. */
   jobCollection = GXGetJobCollection(whichDocument->documentJob);
   /* Set number of copies to 1. */
   copiesInfo.copies = 1;
   err = MyReplaceCollectionItem(&copiesInfo, sizeof(gxCopiesInfo),
            gxCopiesTag, gxPrintingTagID, jobCollection, &oldCopiesInfo,
            &oldCopiesSize);
   nrequire(err, ReplaceCopies_error);
   /* Set page range to "all". */
   pageRangeInfo.simpleRange.optionChosen = gxDefaultPageRange;
   pageRangeInfo.minFromPage = 1;
   pageRangeInfo.simpleRange.fromPage = 1;
   pageRangeInfo.maxToPage = whichDocument->numPages;
   pageRangeInfo.simpleRange.toPage = whichDocument->numPages;
   pageRangeInfo.simpleRange.printAll = true;
```

(continued on next page)

Listing 12. MyPrintOneCopy *(continued)*

```
    err = MyReplaceCollectionItem(&pageRangeInfo,
            sizeof(gxPageRangeInfo), gxPageRangeTag, gxPrintingTagID,
            jobCollection, &oldPageRangeInfo, &oldPageRangeInfoSize);
    nrequire(err, ReplacePageRange_error);
    /* Set destination to "printer". */
    destInfo.toFile = false;
    err = MyReplaceCollectionItem(&destInfo,
            sizeof(gxFileDestinationInfo), gxFileDestinationTag,
            gxPrintingTagID, jobCollection, &oldDestInfo,
            &oldDestInfoSize);
    nrequire(err, ReplaceDestination_error);
    /* Print one copy of our document. */
    err = MyPrintDocument(whichDocument);
    /* Restore original number of copies, page range, and output
       destination in case anybody uses that info. */
ReplaceDestination_error:
    MyReplaceCollectionItem(oldDestInfo, oldDestInfoSize,
       gxFileDestinationTag, gxPrintingTagID, jobCollection, nil, nil);
ReplacePageRange_error:
    MyReplaceCollectionItem(oldPageRangeInfo, oldPageRangeInfoSize,
       gxPageRangeTag, gxPrintingTagID, jobCollection, nil, nil);
ReplaceCopies_error:
    MyReplaceCollectionItem(oldCopiesInfo, oldCopiesSize, gxCopiesTag,
       gxPrintingTagID, jobCollection, nil, nil);
    /* Dispose of pointers that MyReplaceCollectionItem created. */
    if (oldCopiesInfo)
       DisposePtr(oldCopiesInfo);
    if (oldPageRangeInfo)
       DisposePtr(oldPageRangeInfo);
    if (oldDestInfo)
       DisposePtr(oldDestInfo);
    return err;
}
```

The MyReplaceCollectionItem routine, which I created for the Simple Sample GX application, has the format

```
OSErr MyReplaceCollectionItem(void *newData, long collectionItemSize,
                OSType collectionType, long collectionID, Collection
                whichCollection, Ptr *oldData, long *oldDataSize);
```

This routine replaces a collection item and returns a copy of its old data. It takes a pointer to the collection data we want to store, and the size, type, ID, and collection to store it in. In the last two parameters, you pass a reference to a pointer in which to store the existing data and a reference to a long word in which to store its size. If the oldData pointer is nil, the existing data is not returned; otherwise, a new pointer is created in the oldData parameter, and the data is returned there.

MyReplaceCollectionItem allows you to replace a collection item, execute some code, and then restore the collection item. That's exactly what we do in the MyPrintOneCopy routine.

DRAG-AND-DROP PRINTING

The final thing that a QuickDraw GX–aware application should support is the new attribute of the 'pdoc' Apple event, enabling users to print documents by dragging their icons to desktop printers. You need to make only a few changes to your current 'pdoc' Apple event handler, as you can see from Listing 13. With these changes to the Apple event handler, our conversion of the QuickDraw sample application to one that's QuickDraw GX aware is complete.

For information on the 'pdoc' Apple event, see *Inside Macintosh: Interapplication Communication*, Chapter 4.•

Listing 13. 'pdoc' Apple event handler, modified for QuickDraw GX

```
pascal OSErr MyHandlePDOC(AppleEvent *theAppleEvent, AppleEvent *reply,
                  long myRefCon)
{
   OSErr          err;
   AEDescList     docList, dtpList;
   FSSpec         myFSS, dtpFSS;
   long           itemsInList, i;
   AEKeyword      theKeyword;
   DescType       typeCode;
   Boolean        draggedToDTP = false;
   Size           actualSize;
   MyDocumentPtr  newDocument;

   /* See if the document was dragged to a desktop printer. */
   err = AEGetAttributeDesc(theAppleEvent, keyOptionalKeywordAttr,
           typeAEList, &dtpList);
   if (err == noErr) draggedToDTP = true;
   /* If we dragged to a desktop printer, get the name of that
      printer and then throw away the description list for it. */
   if (draggedToDTP) {
      err = AEGetNthPtr(&dtpList, 1, typeFSS, &theKeyword, &typeCode,
               (Ptr) &dtpFSS, sizeof(FSSpec), &actualSize);
      AEDisposeDesc(&dtpList);
   }
   /* Get our document list. */
   err = AEGetParamDesc(theAppleEvent, keyDirectObject, typeAEList,
           &docList);
   nrequire(err, AEError);
   /* Make sure we've accounted for all of the parameters passed,
      and count the number of documents passed in. */
   err = MyCheckAEParams(theAppleEvent);
   nrequire(err, AEError);
   err = AECountItems(&docList, &itemsInList);
   nrequire(err, AEError);
   /* For each entry in the doc list, load it, print it, and close it. */
   for (i = 1; i <= itemsInList; i++) {
      err = AEGetNthPtr(&docList, i, typeFSS, &theKeyword, &typeCode,
               (Ptr) &myFSS, sizeof(FSSpec), &actualSize);
      nrequire(err, AEEntryError);
```

(continued on next page)

Listing 13. 'pdoc' Apple event handler, modified for QuickDraw GX *(continued)*

```
      /* Load the document. */
      err = MyCreateDocument(kDefaultTitle, &newDocument);
      nrequire(err, CreateDocFailed);
      err = MyFSLoadDocument(newDocument, &myFSS);
      nrequire(err, LoadDocFailed);
      /* If we dragged to a desktop printer, select that as the
         output printer for this job, and print one copy. */
      if (draggedToDTP) {
         GXSelectJobOutputPrinter(newDocument->documentJob, dtpFSS.name);
         err = MyPrintOneCopy(newDocument);
      }
      else  /* "Print" chosen from Finder. Show dialog and print. */
         err = MyPrintDocument(newDocument);
      /* Close the document once it's printed. */
LoadDocFailed:
      MyDisposeDocument(newDocument);
   }
   /* When we're all done, throw away the document list and exit. */
CreateDocFailed:
AEEntryError:
   AEDisposeDesc(&docList);
AEError:
   if (gQuitAfterPrinting)
      gQuitting = true;
   return err;
}
```

WHERE TO TAKE IT FROM HERE

Now that you know how to add QuickDraw GX printing to a QuickDraw application, go do it! Think of all the features you'll instantly support by being QuickDraw GX aware. The time hit to gain this level of compatibility is minimal for most applications, and well worth it.

Still not convinced? Take the sample applications and print with both versions under QuickDraw GX. Even this simple program shows that if your applications aren't QuickDraw GX aware, you (and your users) are really missing out.

RELATED READING

- *Inside Macintosh: QuickDraw GX Printing* and *Inside Macintosh: QuickDraw GX Environment and Utilities* (Addison-Wesley, 1994).
- "Getting Started With QuickDraw GX" by Pete ("Luke") Alexander, *develop* Issue 15.
- "Developing QuickDraw GX Printing Extensions" by Sam Weiss, *develop* Issue 15.

Thanks to our technical reviewers Pete ("Luke") Alexander, Hugo Ayala, Tom Dowdy, and Ken Hittleman.•

Making the Most of QuickDraw GX Bitmaps

Besides letting you do a lot of cool things with geometric shapes and typography, QuickDraw GX has useful tools for manipulating bitmaps. For example, bitmap shapes (the QuickDraw GX counterpart to pixMaps) can be skewed, rotated, and scaled, and transforms allow these operations to be performed repeatedly without data loss. Bitmap shapes can share image data, can be used to clip other shapes, and can reside on disk instead of in memory. This article tells how you can use QuickDraw GX to improve the way you handle bitmapped graphics.

DAVID SUROVELL

New users of QuickDraw GX will probably start by going through *Inside Macintosh: QuickDraw GX Objects* or the article "Getting Started With QuickDraw GX" in *develop* Issue 15. If you're mainly a QuickDraw programmer, however, you may have a lot of questions about how QuickDraw GX applies specifically to bitmaps — probably the most commonly used graphic objects. As it turns out, it can do most anything QuickDraw can do, and quite a few useful and exotic new things besides.

If you have at least a nodding familiarity with QuickDraw GX, this article will give useful tips on how to apply your knowledge to bitmap shapes. If you're a QuickDraw GX neophyte, this article will confuse you from time to time, but you may learn enough to decide to make the leap to QuickDraw GX.

CREATING BITMAP SHAPES

It takes about the same information to create a bitmap shape in QuickDraw GX as it does to make a pixMap in QuickDraw. The biggest difference is that while QuickDraw insists that you calculate the size of the image buffer and allocate it explicitly, QuickDraw GX can optionally allocate it for you when the shape is created. This is illustrated in the code in Listing 1, which creates an indexed bitmap shape.

For indexed pixelSize values (1, 2, 4, or 8), you set the gxBitmap's space field to gxIndexedSpace and its set field to a color set (the QuickDraw GX equivalent of a QuickDraw color table) with an appropriate number of entries. Direct pixelSize values (16 or 32) require that the set field be nil. For example, to make the routine in Listing 1 create a 16-bit bitmap shape, you would set the gxBitmap's space field to gxRGB16Space and its set field to nil.

DAVID SUROVELL Where there was once one, there now are three: after approximately 1500 years of bachelorhood, David recently married (Jane) and achieved fatherhood (Elliot Ivan). He once wrote a book on QuickDraw, but that was long ago. When he's not sleeping under his desk at Apple, David's passionate avocations include auditioning as a guitarist for bands that fail to play in public, committing brutal fouls in otherwise friendly soccer matches and basketball games, and playing paintball with other rush-hour commuters.•

Listing 1. Creating an indexed bitmap shape

```
gxShape CreateIndexedBitmapShape(long horiz, long vert,
                                 long targetDepth)
{
   gxBitmap    bitShapeInfo;
   gxColorSet  targetSet;
   gxShape     resultShape;

   if ((horiz <= 0) || (vert <= 0))
      return nil;
   if (targetDepth > 8)
      return nil;

   // Create a familiar "color" gxColorSet.
   // (The default gxColorSet is a gray ramp.)
   targetSet = GetStandardColorSet(targetDepth);
   if (targetSet == nil)
      return nil;

   // Let QDGX calculate the image buffer block size and allocate it.
   bitShapeInfo.image = nil;
   bitShapeInfo.rowBytes = 0;
   bitShapeInfo.width = horiz;
   bitShapeInfo.height = vert;
   bitShapeInfo.pixelSize = targetDepth;
   bitShapeInfo.space = gxIndexedSpace;
   bitShapeInfo.set = targetSet;

   // Use the default color profile.
   bitShapeInfo.profile = nil;
   resultShape = GXNewBitmap(&bitShapeInfo, nil);

   return resultShape;
}
```

Note that the gxBitmap's rowBytes is a long, not a short as in QuickDraw. This means no more convoluted rowByte hacks, no more magic bits needed for flags, and no more unreasonable limits on image width.

Note also that the gxBitmap contains a profile field, a reference to a gxColorProfile (essentially an object with ColorSync data wrapped inside). If this field is nil, QuickDraw GX uses its default profile. Color matching occurs only when the target view port has the gxEnableMatchPort attribute set — by default, it's off.

MANIPULATING BITMAP SHAPES

Once a bitmap shape is created, you can access and change its characteristics with GXGetBitmap and GXSetBitmap.

```
GXGetBitmap(targetShape, &bitmapInfo, &origin);
// Alter the necessary gxBitmap fields here.
. . .
GXSetBitmap(targetShape, &bitmapInfo, &origin);
```

GXSetBitmap is similar to QuickDraw's UpdateGWorld; it lets you change bitmap depth, color specification, and size. To change specific attributes, you may need to modify a combination of fields.

To change a bitmap's width or height, set the width or height field. If QuickDraw GX originally allocated the image buffer, you can set rowBytes to 0 and the image field to nil, and QuickDraw GX will reallocate the buffer. If you allocated the buffer yourself, you'll have to maintain it yourself.

An image isn't scaled when you change size this way. If you increase the width or height, the new areas contain undefined values; if you decrease them, the image is truncated. Bitmap scaling is discussed later in this article. •

To change a bitmap's pixel depth, set the pixelSize field to the desired depth. If the bitmap needs a new color set (which it will, unless the new depth is greater than 8 bits), create it and assign it to the set field. An example that changes the depth to 4-bit is shown in Listing 2.

To change a bitmap's color characteristics, just change the set, space, and profile fields. No changes to pixel data will occur — all pixel values will be interpreted in the new color set. To transform pixel values, you'd need to set up a new bitmap shape and draw the existing bitmap into it. (The offscreen library routine CopyToBitmaps is ideal for this.)

PIXEL VALUE REPRESENTATION

A raster image is, naturally enough, an array of pixel values. For indexed color, each pixel value is an index into an associated color set.

For direct color (16 or 32 bits per pixel), a pixel value is converted directly into a color value by expanding bit fields of the 16- or 32-bit value into three or four 16-bit unsigned integer values.

The expansion of direct pixel values depends on the color space of the raster image and the "packing" of the color components. QuickDraw supports only RGB and a handful of packing schemes, but QuickDraw GX supports a whole family of color spaces and packing formats, some of which are shown in Figure 1.

The packing types are defined in the gxColorSpaces enum in the header file graphics types.h. You'll also find definitions for extended color space specifications, such as gxRGB16Space (gxRGBSpace + gxWord5ColorPacking) and gxARGB32Space (gxLong8ColorPacking + gxRGBASpace + gxAlphaFirstPacking). Only explicitly defined permutations are valid — you can't just make up your own.

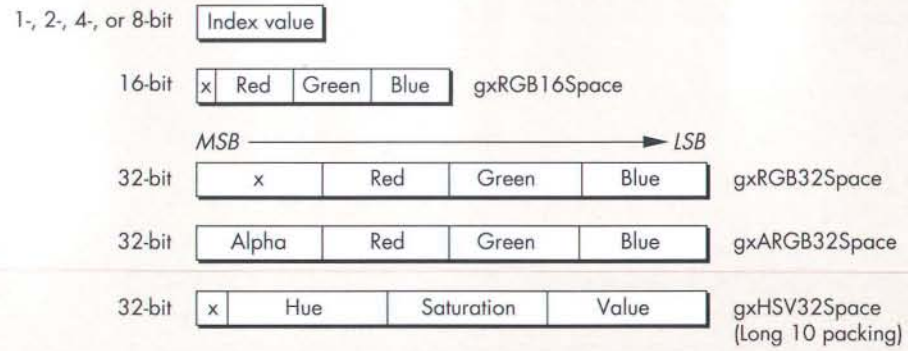


Figure 1. Some QuickDraw GX pixel packing formats

Listing 2. Changing the depth of a bitmap shape

```
void ChangeDepthToFour(gxShape bitmapShape)
{
   gxBitmap imageInfo;

   if ((bitmapShape != nil) &&
         (GXGetShapeType(bitmapShape) == gxBitmapType))
   {
      GXGetBitmap(bitmapShape, &imageInfo, nil);
      if (imageInfo.pixelSize != 4)
      {
         imageInfo.pixelSize = 4;
         imageInfo.space = gxIndexedSpace;
         imageInfo.set = GetStandardColorSet(4);
         GXSetBitmap(bitmapShape, &imageInfo, nil);
      }
   }
}
```

USING DISK-BASED PIXEL IMAGES

QuickDraw GX provides support for disk-based bitmap shapes. They're structurally the same as regular bitmaps, except that their image data is contained in a file, so they're always drawn from disk. Ten calls to GXDrawShape(diskBitmap) means QuickDraw GX reads the entire file from disk ten times. (QuickDraw GX can't assume that you didn't write into the file between accesses.) The idea is that the file system's disk caches will do the work; if the file wasn't changed, subsequent reads should be cached.

Make sure the file size is at least as large as the bitmap, or you'll get an "unexpected end of file" error.•

Disk-based bitmaps have limitations. For one thing, certain routines can't be performed on them — GXSetShapePixel, for example. (See *Inside Macintosh: QuickDraw GX Graphics* for the complete list.) You can't use disk-based bitmap shapes as drawing destinations. If you draw into the data you trigger an error.

So how do you create a disk-based bitmap? As shown in Listing 3, you first set the gxBitmap's image field to gxBitmapFileAliasImageValue. After creating the bitmap shape, create a tag of type gxBitmapFileAliasTagType containing an alias record that references the file containing the target raster data and attach it to the shape.

ACCESSING IMAGE DATA

You can manipulate the image data of bitmap shapes directly. If the image data is maintained by your application, all you have to do is call GXChangedShape afterward. If the image data was allocated by QuickDraw GX, it's more complicated:

1. Force the shape to be heap-resident with GXSetShapeAttributes.
2. Lock the shape with GXLockShape and check for an error.
3. Call GXGetShapeStructure to obtain a reference to the image data.
4. Read from or write to the image data as desired.

Listing 3. Creating a disk-based bitmap

```
gxShape CreateDiskBitmap(FSSpec *fsData, gxBitmap *targetBM)
{
   gxBitmap    localBM;
   gxShape     targetShape;
   gxTag       targetTag;

   if ((fsData == nil) || (targetBM == nil))
      return nil;
   targetShape = nil;
   targetTag = CreateBitmapAliasTag(fsData, 0L);
   if (targetTag != nil)
   {
      localBM = *targetBM;
      localBM.image = gxBitmapFileAliasImageValue;
      targetShape = GXNewBitmap(&localBM, nil);
      if (targetShape != nil)
         GXSetShapeTags(targetShape, gxBitmapFileAliasTagType,
                        1L, -1L, 1L, &targetTag);
      GXDisposeTag(targetTag);
   }
   return targetShape;
}

gxTag CreateBitmapAliasTag(FSSpec *bitmapFS, unsigned long fileOffset)
{
   struct gxBitmapDataSourceAlias  *aliasRecordPtr;
   gxTag          targetTag;
   FSSpec         targetFS;
   AliasHandle    aliasHdl;
   OSErr          iErr;
   long           aliasSize, aliasRecordSize;
   Boolean        wasChanged;

   targetTag = nil;
   aliasHdl = nil;
   aliasRecordPtr = nil;

   // Create an alias and resolve it.
   iErr = NewAlias(nil, bitmapFS, &aliasHdl);
   if (iErr == noErr)
      iErr = ResolveAlias(nil, aliasHdl, &targetFS, &wasChanged);

   // Build up a compact representation for inclusion into a gxTag.
   if (iErr == noErr)
   {
      aliasSize = GetHandleSize((Handle)aliasHdl);
      aliasRecordSize = aliasSize + 2 * sizeof(long);
      aliasRecordPtr = (struct gxBitmapDataSourceAlias*)
                       NewPtr(aliasRecordSize);
      iErr = MemError();
   }
```

(continued on next page)

Listing 3. Creating a disk-based bitmap *(continued)*

```
    // Create the gxTag.
    if (iErr == noErr)
    {
        // Create a gxBitmapDataSourceAlias with specified fileOffset
        // and appropriate aliasRecordSize and aliasRecord.
        aliasRecordPtr->fileOffset = fileOffset;
        aliasRecordPtr->aliasRecordSize = aliasSize;
        BlockMove(*aliasHdl, &aliasRecordPtr->aliasRecord[0], aliasSize);
        targetTag = GXNewTag(gxBitmapFileAliasTagType, aliasRecordSize,
                             aliasRecordPtr);
    }
    // Clean up.
    if (aliasHdl != nil)
        DisposeHandle((Handle)aliasHdl);
    if (aliasRecordPtr != nil)
        DisposePtr((Ptr)aliasRecordPtr);

    return targetTag;
}
```

5. If the image data was changed, call GXChangedShape.
6. Unlock the shape with GXUnlockShape.
7. Call GXSetShapeAttributes to allow the shape to be cached again.

GXLockShape loads an image into memory, so it might not succeed if there isn't enough memory. And don't forget to check a bitmap shape's space field before processing the shape — don't assume that bitmap images are always in RGB space.

See Listing 4 for an example of changing a bitmap shape's data directly.

MEMORY ISSUES

Raster surfers and Photoshop junkies know that raster images can be memory hogs; it's easy to run out of application heap when you allocate them. So what happens when QuickDraw GX runs out of memory? It doesn't. Well, almost never. Here are the steps it will go through, in order, to deliver the memory you need:

1. Flush out-of-date caches.
2. Flush up-to-date caches.
3. If allowed, grow the current gxHeap.
4. Unload shapes and other objects to disk.
5. Give up, and return an error.

Most QuickDraw developers resort to some sort of GrowZoneProc to handle a tight application heap. QuickDraw GX provides a tiered response to abnormal occurrences. Items 1 through 4 above return notices (in the debugging version of QuickDraw GX); item 5 returns an error. All you have to do is implement a routine to handle the notices and errors.

Listing 4. Directly changing an indexed bitmap shape

```
void InvertBitmapShape(gxShape sourceBits)
{
   gxBitmap          sourceInfo, *sourceInfoRef;
   gxShapeAttribute  curAttributes;
   unsigned char     *sourcePtr, *rowPtr;
   long              sourceRowSize, structLen, i, j;
   Boolean           isQDGXImage;

   // Make sure that this is an indexed bitmap shape.
   if (sourceBits == nil)
      return;
   if (GXGetShapeType(sourceBits) != gxBitmapType)
      return;
   GXGetBitmap(sourceBits, &sourceInfo, nil);
   if (sourceInfo.pixelSize > 8)
      return;
   if (sourceInfo.image == gxBitmapFileAliasImageValue)
      return;
   // If the image data was allocated by QuickDraw GX...
   isQDGXImage = (sourceInfo.image == nil);
   if (isQDGXImage)
   {
      // Load and lock the image data.
      curAttributes = GXGetShapeAttributes(sourceBits);
      if (!(curAttributes & gxDirectShape))
         GXSetShapeAttributes(sourceBits,
                              curAttributes | gxDirectShape);
      GXLockShape(sourceBits);
      if (GXGraphicsError(nil) != 0)
         return;

      // Get a reference to the image data.
      sourceInfoRef =
         (gxBitmap*)GXGetShapeStructure(sourceBits, &structLen);
      if ((sourceInfoRef == nil) || (structLen < sizeof(gxBitmap)))
         return;

      sourceInfo = *sourceInfoRef;
   }

   // Invert index values, one row at a time.
   sourcePtr = (unsigned char*)(sourceInfo.image);
   for (i = sourceInfo.height; i > 0; i--)
   {
      rowPtr = sourcePtr;
      sourceRowSize = sourceInfo.rowBytes;
      while (sourceRowSize-- > 0)
      {
         *rowPtr = ~*rowPtr;
         rowPtr++;
      }
```

(continued on next page)

Listing 4. Directly changing an indexed bitmap shape *(continued)*

```
        // Skip to the next row.
        sourcePtr = (unsigned char*)sourcePtr + sourceInfo.rowBytes;
    }

    GXChangedShape(sourceBits);
    if (isQDGXImage)
    {
        GXUnlockShape(sourceBits);
        GXSetShapeAttributes(sourceBits, curAttributes);
    }
}
```

GEOMETRIC OPERATIONS

One of the niftiest features of QuickDraw GX is the ability to perform geometric operations on bitmap shapes. Most of the operators that apply to geometric shapes also apply to bitmaps: rotate, scale, skew, perspective, and clip. In comparison, QuickDraw provides only three geometric operators: scale, clip, and mask.

ALTERING THE TRANSFORM VERSUS THE GEOMETRY

When you change a bitmap shape's geometry (that is, its actual pixel data), whether by rotating, skewing, applying perspective, or scaling, you normally lose image data — it's often impossible to return the image to its pristine state.

You can eliminate this data loss by instead applying geometric operators to a shape's *transform*. A shape can make use of a 3 x 3 matrix to mathematically change its appearance when rendered without changing the underlying data. This is especially important for bitmaps. Figure 2 shows both possibilities of multiple rotations of a bitmap.

Figure 2. Successive rotations of a bitmap

Rotation, translation (change in origin), skew, perspective, and scale operations can all be performed on transforms directly, by GXRotateTransform, GXSkewTransform, and so forth, or indirectly, using the gxMapTransformShape attribute.

When a shape's gxMapTransformShape attribute is set, geometric operations automatically apply to its transform rather than its geometry. Bitmap and picture shapes default to having this attribute set; other shapes begin with it off. This means that if you convert a polygon shape (for example) to a bitmap shape, the gxMapTransformShape attribute won't automatically be set.

When a QuickDraw GX routine modifies a bitmap shape's geometry, a clip shape is often attached to define the geometric extent of the modified bitmap. More often

than not, the bitmap's image buffer is expanded, as shown in Figure 3. Rotating a bitmap's geometry can increase its memory requirements by over 40%.

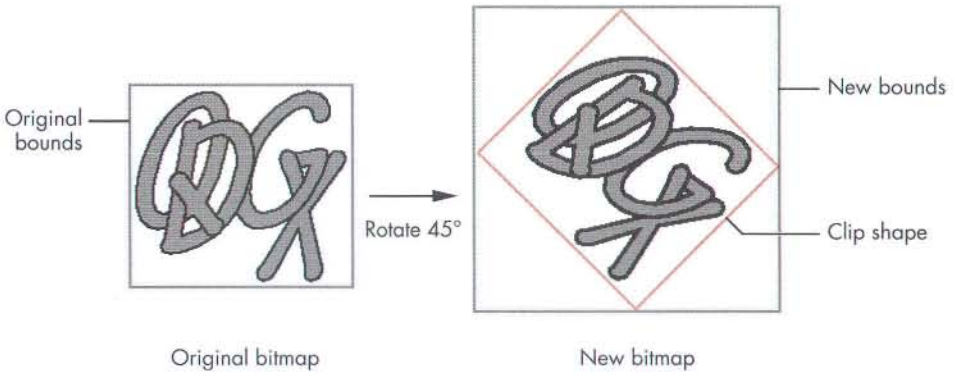


Figure 3. Effect of GXRotateShape on bitmap geometry

ROTATION

There aren't many QuickDraw programmers who haven't wished for a simple way to rotate bitmaps. GXRotateShape takes parameters for the target shape, degrees clockwise to rotate, and center point of rotation, as shown in Listing 5.

Listing 5. Rotating a bitmap shape

```
void RotateBitmap(gxShape targetShape, Fixed theta)
{
   gxBitmap    targetBM;
   gxPoint     origin, shCenter;

   // Determine the bitmap shape's current center point.
   GXGetBitmap(targetShape, &targetBM, &origin);
   shCenter.x = ff(targetBM.width) / 2 + origin.x;
   shCenter.y = ff(targetBM.height) / 2 + origin.y;

   // Rotate it around its center point.
   GXRotateShape(targetShape, theta, shCenter.x, shCenter.y);
}
```

SKEWING AND PERSPECTIVE

Skewing and perspective are just as much fun as rotation, and even more useful as general-purpose graphic effects. The code in Listing 6 illustrates a simple type of perspective; Figure 4 shows the results of this perspective mapping.

SCALING

You can expand or shrink bitmap shapes, like other shape types, with GXScaleShape. QuickDraw pixMaps are scaled by setting the destination rectangle passed to CopyBits, whereas GXScaleShape uses a scaling factor. To convert your QuickDraw bitmap scaling code into the equivalent QuickDraw GX code, you have to calculate this scaling factor. Listing 7 shows how.

You can flip a bitmap horizontally or vertically by using negative scaling values. •

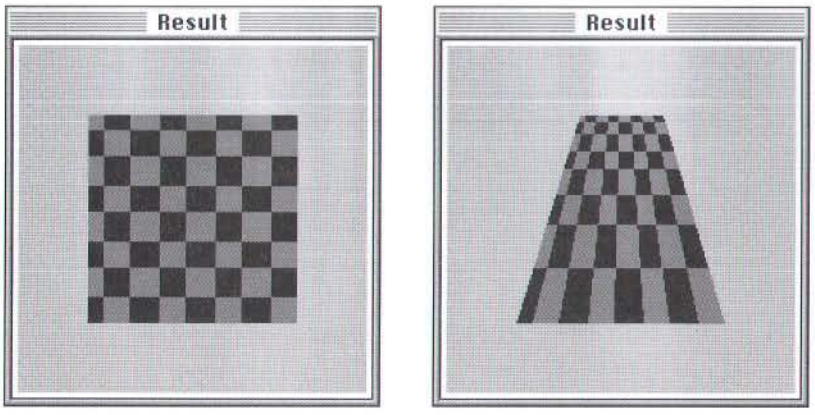
Listing 6. Applying perspective to a bitmap shape

```
void TrapezoidalWarp(void)
{
   gxShape  bitsShape, warpShape;
   long     trapezoidData[] =
   {
      1L, 4L,
      ff(130), ff(100), ff(170), ff(100),
      ff(200), ff(200), ff(100), ff(200)
   };

   bitsShape = CreateBasicBitmapShape();
   warpShape = GXNewShapeVector(gxPolygonType, trapezoidData);
   if (warpShape != nil)
   {
      ShapeSetPolyMap(bitsShape, warpShape);
      GXDisposeShape(warpShape);
   }
   GXDrawShape(bitsShape);
}

void ShapeSetPolyMap(gxShape targetShape, gxShape mappingShape)
{
   gxRectangle    boundsRect;
   gxPolygon      *mapPoly, *targetPoly;
   gxMapping      theMapping;
   gxShape        targetBounds;
   long           ignored;

   if (targetShape == nil)
      return;
   if ((mappingShape == nil)
         || (GXGetShapeType(mappingShape) != gxPolygonType))
      return;

   // Determine the dimensions of the target shape.
   GXGetShapeBounds(targetShape, 0L, &boundsRect);
   targetBounds = GXNewRectangle(&boundsRect);
   if (targetBounds == nil)
      return;

   // Scale the mapping shape to the dimensions of the target shape.
   GXSetShapeBounds(mappingShape, &boundsRect);
   GXSetShapeType(targetBounds, gxPolygonType);

   // Load & lock both shapes so that their structures can be accessed.
   GXSetShapeAttributes(mappingShape,
               GXGetShapeAttributes(mappingShape) | gxDirectShape);
   GXLockShape(mappingShape);
   GXSetShapeAttributes(targetBounds,
               GXGetShapeAttributes(targetBounds) | gxDirectShape);
   GXLockShape(targetBounds);
```

(continued on next page)

Listing 6. Applying perspective to a bitmap shape *(continued)*

```
    // NOTE: Structure is actually of type gxPolygon.
    mapPoly = (gxPolygon*)GXGetShapeStructure(mappingShape, &ignored);
    targetPoly = (gxPolygon*)GXGetShapeStructure(targetBounds, &ignored);

    if ((mapPoly != nil) && (targetPoly != nil))
    {
        // Skip past the gxPolygons contour count to the first contour.
        mapPoly = (gxPolygon*)((Ptr)mapPoly + sizeof(long));
        targetPoly = (gxPolygon*)((Ptr)targetPoly + sizeof(long));

        // Calculate the desired shape mapping.
        // PolyToPolyMap() is in "mapping library.c."
        PolyToPolyMap(targetPoly, mapPoly, &theMapping);
    }

    // Release both shapes from bondage.
    GXUnlockShape(mappingShape);
    GXSetShapeAttributes(mappingShape,
                GXGetShapeAttributes(mappingShape) & ~gxDirectShape);
    GXUnlockShape(targetBounds);
    GXSetShapeAttributes(targetBounds,
                GXGetShapeAttributes(targetBounds) & ~gxDirectShape);

    // Set the target shape's mapping as desired.
    GXSetShapeMapping(targetShape, &theMapping);

    GXDisposeShape(targetBounds);
}
```



Before perspective

After perspective

Figure 4. Applying perspective to a bitmap shape

CLIPPING AND MASKING

QuickDraw GX can do some neat tricks with clipping. These tricks work with bitmap shapes, too. For example, to create a gradient-filled polygon, you can make a rectangular bitmap shape with a gradient and then set the polygon shape as the bitmap's clip shape. (For another example, see Graphical Truffles in this issue.)

Listing 7. Calculating a scaling factor

```
void BitmapShapeScaleQDStyle(gxShape targetShape, Rect *qdSourceR,
                             Rect *qdDestR)
{
   gxPoint  centerPt;
   fixed    scaleFactorH, scaleFactorV;

   scaleFactorH = FixRatio(qdSourceR.right - qdSourceR.left,
                           qdDestR.right - qdDestR.left);
   scaleFactorV = FixRatio(qdSourceR.bottom - qdSourceR.top,
                           qdDestR.bottom - qdDestR.top);
   centerPt.x = ff((qdDestR.right + qdDestR.left) / 2);
   centerPt.y = ff((qdDestR.bottom + qdDestR.top) / 2);
   GXScaleShape(targetShape, scaleFactorH, scaleFactorV, centerPt.x,
                           centerPt.y);
   GXMoveShapeTo(targetShape, ff(qdDestR.left), ff(qdDestR.top));
   GXDrawShape(targetShape);
}
```

You can use 1-bit bitmap shapes as clip shapes, too. The effect is just like that of CopyMask; pixels in the source shape are drawn only where the clipping bitmap pixel value is nonzero. (On this issue's CD, you'll also find example code that does image processing similar to CopyDeepMask using the new transfer modes.)

Clipping occurs in geometry space, before transform mapping, so a bitmap's clip shape should be based on its bounds rectangle, not its rendered location.•

To convert geometric shapes into masking bitmap shapes, you can call the GXSetShapeType routine to convert the shape to a 1-bit mask bitmap.

With GXCheckBitmapColor, you can generate a masking bitmap from an existing bitmap shape. If you pass GXCheckBitmapColor a color set, it puts 0 in the result bitmap for source pixel values that are in the color set. If you pass it a color profile, it puts 0 in the result bitmap for source pixel values that are within the color profile's gamut. The result bitmap can be useful for color correction.

QUICKDRAW GX TRICKS FOR QUICKDRAW DOGS

QuickDraw GX has ways to do almost anything you can do with QuickDraw. All you need to know is how their environments and feature sets compare, and you'll understand how to convert from one to the other.

THE VIEW PORT LIST VERSUS THE GRAPHICS PORT

Most of the time you won't have to concern yourself with view ports at rendering time, because there's no sense of the "current port" as there is in QuickDraw. Here's the recommended method for drawing an existing shape into a new view port:

1. Copy the shape's transform and install the desired destination view port into the copy.
2. Call GXDrawShape.
3. Restore the original transform.
4. Dispose of the copied transform.

Examples of preserving view port lists can be found in the library routine CopyToBitmaps and in the DrawShapeOffscreen example later in this article (Listing 9).

BITMAPS AND TRANSFER MODES

QuickDraw GX has a lot of transfer modes. This is a good thing, really. Not all transfer modes are functionally equivalent to those in QuickDraw, but the transferMode library is fairly complete. Many of the capabilities of QuickDraw search procedures can be implemented using transfer modes. (The first page of *Inside Macintosh: QuickDraw GX Graphics* has color pictures of the new transfer modes in action.)

The transfer mode is contained in a shape's ink. Since transfer modes are applied on a per-component basis, you can easily get some groovy effects. For example, you can add the hue of one image to the brightness of another. Usually, though, you'll want all components to use the same mode. The transferMode library routine SetCommonTransfer will do this for you.

There are some differences between QuickDraw GX transfer modes and those found in QuickDraw:

- Dithering is a view port feature, not a transfer mode. Halftoning is also available on a per-gxViewPort basis. These two features are mutually exclusive; you can't dither and halftone at the same time.
- Transparency is not a single mode. It's a whole family of modes based on alpha component values.
- All QuickDraw GX transfer modes occur in color space, while some QuickDraw transfer modes are bitwise.

ONSCREEN BITMAPS

QuickDraw GX maintains a view device list that mirrors the QuickDraw GDevice list. (Utility routines are provided for getting one if you have the other.) The Window Manager is patched in a couple of places so that a window's view port transforms and image memory are maintained when it enters and leaves GDevice real estate.

Drawing a bitmap onscreen obeys the screen GDevice's index entry protections — QuickDraw GX doesn't use indexes reserved by the Palette Manager for other applications. If you want to draw an image that uses animated palette entries, you'll need to clone references to the destination viewDevice color set and profile, and then insert those references into the bitmap shape before drawing. Example code that does this is on this issue's CD.

COPYBITS IN QUICKDRAW GX

Let's see what it takes to make GXDrawShape do what CopyBits does. CopyBits has several explicit parameters: the source, destination, clipping region, and transfer mode. In QuickDraw GX, the source is the bitmap shape. The destination is defined by the shape's view port list. The clipping region is any shape that you attach to the bitmap shape with GXSetShapeClip. As mentioned before, the transfer mode is contained in the shape's ink.

So, to do a CopyBits-style blit in QuickDraw GX:

1. Set up the shape's view port list.
2. Determine the transfer mode (usually just "copy," but it's your choice).

3. Adjust the shape clip. Don't change the device clip or view port clip.
4. Adjust the transform if you want to reposition, scale, skew, rotate, or apply perspective to the shape.
5. Call GXDrawShape.
6. Clean up as needed.

QuickDraw GX doesn't implement all of the color capabilities of CopyBits. There's no colorizing and no color interpolation for indexed values beyond the end of a bitmap's color set.•

DRAWING OFFSCREEN WITH QUICKDRAW GX

Successive QuickDraw implementations have presented newer and better ways to draw into a offscreen image buffer. The QuickDraw GX offscreen library contains routines to help maintain the data structures necessary to implement the equivalent of a GWorld.

The example in Listing 8 uses the CreateIndexedBitmapShape routine from Listing 1 and the library routine CreateOffscreen to create a fully functional offscreen bitmap.

Listing 8. Creating an offscreen bitmap

```
OSErr MakeIndexedOffscreen(offscreen *targetOffWorld, long horiz,
                           long vert, long targetDepth)
{
   gxShape  bitsShape;

   if (!CheckArguments(...))
      return paramErr;
   bitsShape = CreateIndexedBitmapShape(horiz, vert, targetDepth);
   if (bitsShape == nil)
      return paramErr;
   CreateOffscreen(targetOffWorld, bitsShape);
   return noErr;
}
```

You might think drawing into a QuickDraw GX offscreen bitmap would be difficult, but it's not. To draw a shape into the offscreen bitmap, set its view port list to the offscreen bitmap's view port and call GXDrawShape (see Listing 9).

BITMAP SHAPES VERSUS PIXMAPS

Sometimes, converting existing QuickDraw code to QuickDraw GX is impractical. If your application needs to use the same data in both offscreen pixMaps and bitmap shapes, it can, provided that the bitmap shape is packed the same as the pixMap — that is, of identical width, height, pixel depth, and color space.

To use bitmap shape data in a QuickDraw pixMap, build the pixMap with the baseAddr the same as the gxBitmap.image. (Make sure that the bitmap shape is locked down.) To use pixMap data in QuickDraw GX, create a gxBitmap with the image field set to the base address of the source pixMap.

Listing 9. Drawing into an offscreen bitmap

```
void DrawShapeOffscreen(offscreen *offGXWorld, gxShape targetShape)
{
   gxTransform newXform, savedXform;

   if ((offGXWorld == nil) || (targetShape == nil))
      return;
   if (offGXWorld->port == nil)
      return;

   savedXform = GXGetShapeTransform(targetShape);
   newXform = GXCopyToTransform(nil, savedXform);
   GXSetTransformViewPorts(newXform, 1L, &(offGXWorld->port));
   GXSetShapeTransform(targetShape, newXform);

   GXDrawShape(targetShape);

   GXSetShapeTransform(targetShape, savedXform);
   GXDisposeTransform(newXform);
}
```

THE QUICKDRAW GX LIBRARIES

Several libraries are included with the QuickDraw GX Software Developer's Kit. They contain, among other things, routines for offscreen rendering and converting image data between QuickDraw and QuickDraw GX. The library code instructs by example and is a good starting point for your own library.

The library code is not completely tested. You should treat it as template code, not a final solution.•

The offscreen library. This library contains support for offscreen bitmaps, copying between bitmap shapes, and simple gradient fills. The offscreen image implementation is basic but solid (it lacks some of the features found in QuickDraw GWorlds, such as automatic longword realignment of images). The utility routine CopyToBitmaps is also useful; it shows a good example of saving a view port list.

The math library. This library contains a number of useful routines for manipulating mappings. The routine PolyToPolyMap is used in the trapezoidal warp example (Listing 6). The header file math routine.h contains essential macros for conversion between fixed-point, floating-point, and integral values.

The ramp library. Get your gradient fills here. Pleasing to the eye, easy on the code. A gradient-filled bitmap can be rotated and clipped, and voilà! Gradient-filled shapes.

The qd and oval libraries. The qd library has facilities for conversion of bitmap and color data between QuickDraw and QuickDraw GX formats. The oval library has real ovals, not those phony squished QuickDraw things.

The transferMode library. This library facilitates access to a shape's transfer mode information and contains routines for emulating most of the QuickDraw transfer modes. It also contains a bonus — one of my favorite routines. If you've ever wanted

to get the results of a QuickDraw transfer mode on color values without having to use CopyBits, TransmogrifyColor is for you. Check it out.

The storage library. This library implements spooling routines for use with GXFlattenShape and GXUnflattenShape, which you'll need for reading and writing shapes to and from files. These routines detect errors but don't report them, so they're only useful as templates.

The camera library. Perspective is cool, but hard to use unless your math skills are well developed. This library provides nifty 3-D techniques.

AND A FEW MORE THINGS . . .

Here I'll point out some caveats and additional interesting features of QuickDraw GX, just so you know what to look for (and look out for).

EXECUTION OVERHEAD

How fast are QuickDraw GX blits? How slow does an offscreen, 256 x 256, 45°-rotated, 32-bit, YXY, gradient-filled bitmap draw into a window on a 4-bit monitor? How much for all of these shiny pebbles? It depends. Let's look at the issues involved. QuickDraw GX and QuickDraw have much in common here:

- They're fastest when there's no conversion of value or image location.
- Common code paths are optimized inside the API: 8-bit to 8-bit, 1-bit to 1-bit, 24-bit to 8-bit, no clipping, rectangle clipped.
- Blits involving complex transformations are usually orders of magnitude slower.

Some transformations require more processing. QuickDraw GX does only as much work as the transformation matrix mandates. From fastest to slowest, the order is: no transformation (or translation only); scaling; skewing or rotation; perspective. •

The basic performance guidelines are similar to automotive fuel efficiency ratings — though we have no hard estimates, mileage is better on a smooth highway (no color mapping, skewing, or scaling) than on surface streets.

A transform mutation can require a 3 x 3 matrix operation for each pixel value when rendered. That's a lot of fixed-point multiplications. If execution speed is critical and the mutated version will be used a lot, copy the bitmap shape, mutate the geometry, and draw like crazy. Otherwise, mutate the transform and draw as needed.

SHARED IMAGE BUFFERS

A bitmap shape's raster image buffer can be shared by other bitmap shapes. Just make the source bitmap shape's image field the same as that of another bitmap shape. GXCopyToShape uses this sharing of image buffers. If you need a copy of a bitmap shape (or a picture that contains bitmap shapes) to have its own image buffer, use GXCopyDeepToShape.

USING BITMAPS AS PATTERNS

Bitmap shapes can be used as patterns. Unlike QuickDraw, QuickDraw GX has no limitation on area dimension or size of raster data in a pattern. To do simple tiling, you can just set the bitmap pattern on the shape.

You can align the pattern to all destination view ports simply by setting the gxPortAlignPattern attribute. This forces all shapes drawn with that pattern in a given view port to visually line up with each other. Another pattern attribute, gxPortMapPattern, keeps a pattern from being affected by a shape's transform; this is useful, for example, when you want a shape rotated and its pattern unrotated.

BITMAP SHAPE EQUIVALENCE

You can test QuickDraw GX shapes for equivalence by calling GXEqualShape. However, this routine doesn't account for mapping effects. For example, a bitmap gradient from black to white would be considered not equal to a white-to-black gradient bitmap whose transform is rotated 180°, even though the two shapes would produce identical results when drawn.

SIMPLIFICATION

GXSimplifyShape reduces an indexed bitmap to its simplest representation, even reducing the pixel depth when possible. For example, if an 8-bit-deep bitmap shape contains only 15 colors, GXSimplifyShape will convert it to a 4-bit-deep bitmap. If a bitmap is all one color, it will be converted into a rectangle shape — it won't be a bitmap shape any more.

SUBSET EDITING

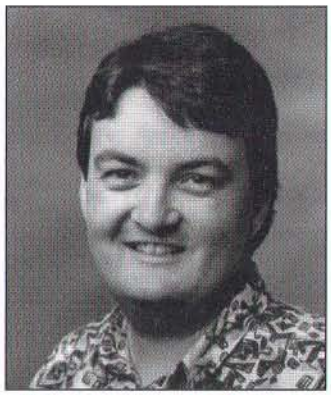
QuickDraw GX provides tools for working with area subsets of bitmaps. A piece can be copied from a source bitmap via GXGetBitmapParts, edited, and then blasted back into the source image with GXSetBitmapParts.

Individual pixel values can be accessed with the GXGetShapePixel and GXSetShapePixel routines. Unlike in QuickDraw, these routines don't need to reference a gxViewDevice to determine the color.

SO GET GOING

As you can see, QuickDraw GX does some really cool things with bitmaps. The transforms alone make it worthwhile — it's easy to get addicted to rotating and skewing your bitmaps without having to do a lot of work. The new transfer modes are great. All the rest is a bonus. In the future, when memory is cheap and every machine is fast, you'll see more and more Macintosh systems and applications become dependent on QuickDraw GX.

Thanks to our technical reviewers Pete ("Luke") Alexander, Josh Horwich, and Chris Yerga.•



GRAPHICAL TRUFFLES

A Cool QuickDraw GX Clipping Effect

PETE ("LUKE") ALEXANDER

Wading through the vast sea of documentation you get with the QuickDraw GX Software Developer's Kit, you may not realize all the cool things this new graphics model lets you do. In this column, we'll look at just one of those cool things: clipping one shape to the inside of another arbitrary shape. As an illustration, we'll take a bitmapped image of a tropical beach (Figure 1) and clip it to the word "BAY" (Figure 2). Figure 3 shows the result. There's a sample application and source code on this issue's CD.

CREATING THE SHAPES

We'll be collecting the results of our operations into a picture shape. We'll assume that the QuickDraw GX environment has already been set up and that we have a window ready to draw into.

For information on setting up the QuickDraw GX environment, see "Getting Started With QuickDraw GX" in *develop* Issue 15. The full details are in *Inside Macintosh: QuickDraw GX Objects* and *Inside Macintosh: QuickDraw GX Graphics.*•

We start by creating an empty picture to which we can add shapes:

```
gxShape      thePicture;
thePicture = GXNewShape(gxPictureType);
```

The bitmap shape we'll be clipping (Figure 1) is stored in our application's resource fork as a 'pxmp' resource. We retrieve it with the GetPixMapShape call from the qd library.

Figure 1. The shape to be clipped

Figure 2. The clip shape

Figure 3. Result of clipping Figure 1 to Figure 2

```
gxShape      theBitmap;
theBitmap = GetPixMapShape(kPixMapID);
```

To create the clip shape, we start with a text shape containing the word BAY.

PETE ("LUKE") ALEXANDER Regular readers of Luke's columns will know that he took his sabbatical from Apple this summer. He sent us a postcard: "Hey dudes! While you're looking out your windows at Silicon Valley smog, I'm looking out my windows at the rocky faces of Montana. Have you heard of National Parks? They're places of unusual beauty set apart from development. Not that kind of development! The kind that causes smog. At Yellowstone Park I enjoyed the clear air, I found where the buffalo roam, and I learned why they call it Old Faithful. I'm now flying somewhere over Montana (don't worry, I'm steering with my knee), admiring the mountains with their lingering snow and enjoying the wide open spaces. Oops, mountain ahead and no more room on the card." He signed it, "See you later, Luke" but then crossed out the "See you later." We're worried, really worried.•

```
gxShape     theClip;
theClip = GXNewText(3, (unsigned char*)"BAY", nil);
```

POSITIONING AND SCALING THE TEXT SHAPE

Before we set our text shape to be the clip of our bitmap shape, we need to move it to the top left corner of the bitmap shape and then scale it to encompass the entire bitmap.

We'll look at two ways to do this. The first method takes us through all the steps in the process, while the second is simpler and lets QuickDraw GX do more of the work for us.

The origin of a text shape is on the baseline, but in this case, because there are no descenders in the text, we just use the bottom left corner of the shape's bounds. We need to offset the text shape by its own height from the top of the bitmap shape. So we need to know the bounds of both the bitmap shape (to find the coordinates of its top left corner) and the text shape (to calculate its height):

```
gxRectangle  bitmapBounds, textBounds;
GXGetShapeBounds(theBitmap, 0, &bitmapBounds);
GXGetShapeBounds(theClip, 0, &textBounds);
textHeight = textBounds.bottom - textBounds.top;
GXMoveShapeTo(theClip, bitmapBounds.left,
          bitmapBounds.top + textHeight);
```

Our original text shape, which is the default text size of 12 points, isn't big enough to cover the entire bitmap shape. If we were to do the clipping at this point, we would get just a tiny piece of the corner instead of the whole bitmap. To get the whole thing, we have to scale the text shape to match the size of the bitmap shape.

For the text shape to scale linearly, without taking the font's hinting into account, we have to turn off QuickDraw GX's built-in metrics and contour grid-fitting capabilities. We do this by setting the shape's text attributes as follows:

```
GXSetShapeTextAttributes(theClip,
   gxNoMetricsGridText | gxNoContourGridText);
```

Although QuickDraw GX lets us clip to any arbitrary shape, clipping is actually limited to primitive shapes only. That is, the clip shape must be a shape whose geometry and fill properties by themselves define the shape; primitive shapes don't use information from a style or transform object. This is not a problem, though, because we can easily convert a shape to its primitive form. The following call does the job:

```
GXPrimitiveShape(theClip);
```

The result is a filled path shape (which is a primitive shape) representing the outlines of the letters in the word BAY.

Next we need to determine how much to scale the text shape to encompass the entire bitmap shape. We do this by finding the width and height of both shapes' bounds rectangles and scaling the text shape in each direction by the ratio between the two:

```
Fixed     bitmapWidth, bitmapHeight;
Fixed     textWidth, textHeight;
Fixed     xScale, yScale;

// Determine the width ratio.
bitmapWidth = bitmapBounds.right -
              bitmapBounds.left;
textWidth = textBounds.right - textBounds.left;
xScale = FixedDivide(bitmapWidth, textWidth);

// Determine the height ratio.
bitmapHeight = bitmapBounds.bottom -
               bitmapBounds.top;
textHeight = textBounds.bottom - textBounds.top;
yScale = FixedDivide(bitmapHeight, textHeight);

GXScaleShape(theClip, xScale, yScale,
         bitmapBounds.left, bitmapBounds.top);
```

We've now scaled the text shape to encompass the entire area of our bitmap shape, and we're ready to use it to clip the bitmap shape. But it turns out that there's a shorter way to do this; we can actually accomplish the same thing with only three lines of code:

```
GXSetShapeTextAttributes(theClip,
   gxNoMetricsGridText | gxNoContourGridText);
GXGetShapeBounds(theBitmap, 0, &textBounds);
GXSetShapeBounds(theClip, &textBounds);
```

As in the earlier method, we turn hinting off and get the bounds of our bitmap shape. The magic here is in the GXSetShapeBounds call: when applied to typographic shapes, this call positions and scales the text to the new bounds and converts it to a primitive shape. That's it! QuickDraw GX does the hard work for us automatically, and we're ready to set the clip of our bitmap shape.

SETTING THE CLIP SHAPE

Now that the text shape is positioned and scaled the way we want it, we're ready to set it up as the clip of our bitmap shape:

```
GXSetShapeClip(theBitmap, theClip);
```

This call changes the clip shape contained within the transform used by our bitmap shape, replacing it with the new clip shape that we've created.

If the same transform were being shared by other shapes, QuickDraw GX would make a copy of the transform to associate with our bitmap shape, ensuring that the new clip would not affect any of the other shapes sharing the same transform.•

The next step is to add the clipped bitmap shape to our picture with GXSetPictureParts. (Note that we could also use the library call AddToShape or the core call GXSetShapeParts to do the same thing, but I chose to be more explicit here.)

```
GXSetPictureParts(thePicture, 0, 0, 1, &theBitmap,
                  nil, nil, nil);
```

The picture now contains a new reference to the original bitmap shape, complete with the new clip we've added to it. Once there's a reference to it in the picture, we don't need the original reference anymore, so we dispose of it:

```
GXDisposeShape(theBitmap);
```

DRAWING THE OUTLINE

The last order of business is to draw an outline around the clipped bitmap. We can accomplish this by setting up the drawing characteristics of our scaled text shape to draw the outline and then adding it to our picture. To frame the shape with a closed outline, we set its fill characteristic to gxClosedFrameFill. Since we want the frame to lie outside the geometry of the shape, we set its style attributes to gxOutsideFrameStyle. Finally, we set the pen size to 3 to get a satisfyingly fat outline, and we set the color of the outline to blue:

```
GXSetShapeFill(theClip, gxClosedFrameFill);
GXSetShapeStyleAttributes(theClip,
                          gxOutsideFrameStyle);
GXSetShapePen(theClip, ff(3));
SetShapeCommonColor(theClip, blue);
```

Now we can add the outlined shape to our picture and dispose of it, as before:

```
GXSetPictureParts(thePicture, 0, 0, 1, &theClip,
                  nil, nil, nil);
GXDisposeShape(theClip);
```

DRAWING THE PICTURE

Before drawing our picture, we'll move it down a little from the top left corner of the window. We could do this by retrieving the picture's transform and shifting it down and to the right with GXMoveTransform; however, since the gxMapTransformShape attribute is set for pictures by default, we can just call GXMoveShape:

```
GXMoveShape(thePicture, f(20), ff(15));
```

Finally, we're ready to draw the picture:

```
GXDrawShape(thePicture);
```

We're done! The result should look like Figure 3.

THAT'S ALL, FOLKS

You've had a quick look at one of the many cool things you can do with the QuickDraw GX graphics system. As you can see from this example, the power and flexibility of QuickDraw GX can give your application the ability to do things you could only dream about until now.

Thanks to Hugo Ayala and Cary Clark for reviewing this column.•

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Let us know what you think.

Send editorial suggestions or comments to AppleLink DEVELOP or to:

Caroline Rose
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)974-6395

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: JOHNSON.DK
Internet: dkj@apple.com
CompuServe: 75300,715
Fax: (408)974-6395

Pick Your Picker With Color Picker 2.0

The limitations of the old Color Picker Package forced many developers to write their own color pickers. The flexibility of Color Picker version 2.0 overcomes the old limitations and provides many new features — most notably, use with ColorSync color. Now it's easy to design color pickers to suit your needs. This article describes how to use the new Color Picker Manager and take advantage of its customization features from within your application.

SHANNON HOLLAND

Apple designed the Color Picker Package as a way for applications to present a standard user interface for color selection. The goal in developing Color Picker version 2.0 was to remain compatible with the existing Color Picker Package while providing tighter integration of color pickers with the application and allowing development of customized color pickers (for example, to support other color spaces or specific devices).

These goals were achieved by adding a Color Picker Manager, turning color pickers into components, and separating the color picker components from the Color Picker Manager. As components, color pickers are now accessed through the Component Manager, which provides a layer between the application and the color picker component. In other words, the application calls the Color Picker Manager, which then calls the Component Manager, which calls the color picker component. In the old Color Picker Package, the application called the color picker directly.

This separation of the color picker components from the Color Picker Manager allows new color picker components to be dynamically added to the system by the user or an application. Once a new color picker component has been registered to the Component Manager, it's available for use by the Color Picker Manager.

The interface to the new Color Picker Manager is divided into high- and low-level calls:

- The high-level calls are designed to be used with a minimum of fuss, but provide access to nearly the whole feature set available to the application through the Color Picker Manager. For compatibility with previous versions, the old high-level call,

SHANNON HOLLAND, once of Apple and now starting up something elsewhere, had little time to write his bio for this article. His only three activities include working, eating, and sleeping. Once upon a time he had a life in which he enjoyed photography, cultural activities, and abusing his friends.

Color Picker version 2.0 ships with QuickDraw GX and also with System 7.5. The forthcoming *Inside Macintosh: Advanced Color Imaging* will describe Color Picker 2.0 in detail.

GetColor, is still there. A new high-level call, PickColor, replaces GetColor and offers a much broader feature set.

- The low-level calls are designed to allow maximum flexibility. They let the application determine the type of dialog the color picker is placed in, rather than using the modal dialog you get with high-level calls. The application can also set the current color and maintain explicit control over the event loop. Color pickers that are invoked through the low-level calls can exist for the life of an application.

This article discusses how to use these calls and take advantage of the new Color Picker Manager. The code examples are provided on this issue's CD. Color Picker 2.0 allows multiple color picker components to exist on a system at one time (through the Component Manager). Although the interface for these components is public, this article doesn't discuss the creation of color picker components.

SPECIFYING COLORS

Unlike the old Color Picker Package, Color Picker 2.0 uses the more complete ColorSync definition of a color, which contains both a color and a profile. The profile defines the color space of the color (which includes the type of color — CMYK, HSL, RGB, and so on). You can also specify a destination profile, which describes the color space of the device for which the color is being chosen (for example, a color printer that will eventually print the document). Given knowledge of the destination profile, color pickers that are ColorSync aware can help the user choose a color that's within the destination device's gamut.

ColorSync is described in the forthcoming *Inside Macintosh: Advanced Color Imaging*. See also "Print Hints: Syncing Up With ColorSync" in *develop* Issue 14.•

The ColorSync definition for a color, shown below, is used only with the new calls. The old call, GetColor, still uses RGBColor for compatibility. These structures are compatible with QuickDraw GX.

```
typedef struct CMProfile **CMProfileHandle;

typedef union {
   RGBColor       rgb;
   unsigned short reserved[4];
} CMColor, *CMColorList;

typedef struct PMColor {
      CMProfileHandle   profile;
      CMColor           color;
} PMColor, *PMColorPtr;
```

If you're specifying an RGB color with no particular profile, you can simply set the CMProfileHandle field of PMColor to nil, which uses the system profile. To specify a color that uses a profile, you need to provide the profile that describes that color.

USING THE HIGH-LEVEL CALLS

The high-level calls are designed to handle the most common uses for the Color Picker Manager. The old GetColor call provides access to the new dialog and the color picker component, but not to any of the new features that are accessible through the Color Picker Manager (such as ColorSync colors).

The new PickColor call is designed to replace GetColor. It can be used very simply, providing roughly the same feature set as GetColor, or it can be used to take advantage of some of the more advanced features of Color Picker 2.0.

The new dialog for the high-level calls is much the same as the old one. A new button, More Choices, reveals a list of all available color pickers (and changes to "Fewer Choices"; see Figure 1). Clicking a color picker in the list makes it the current color picker for the dialog. Both PickColor and GetColor display this dialog.

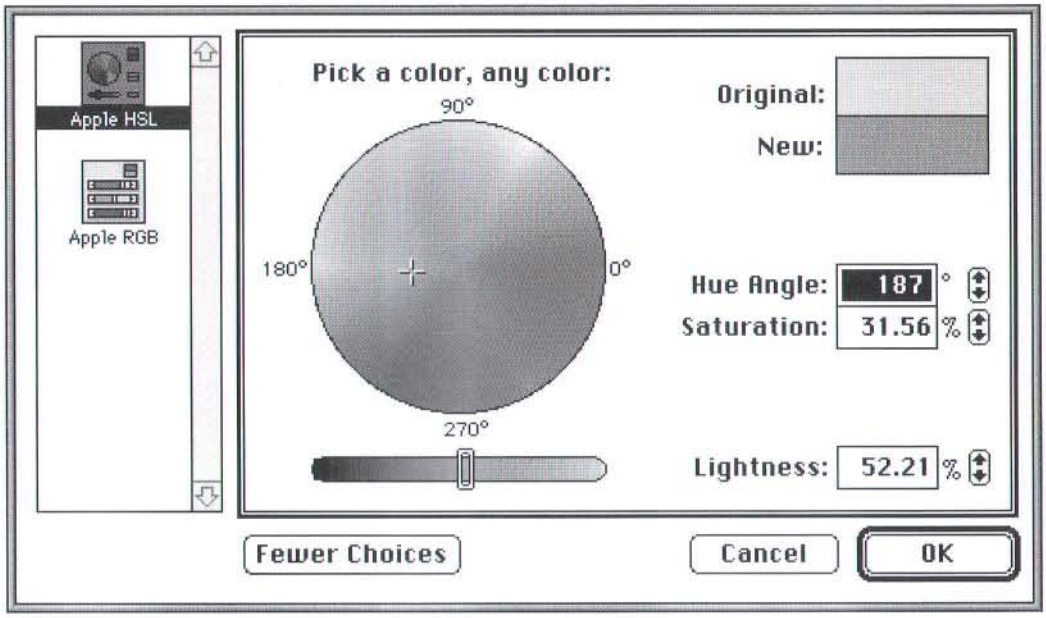


Figure 1. Color picker dialog for high-level calls

The biggest difference between PickColor and GetColor is that PickColor allows the application to provide a pointer to an event filter procedure. If the application supplies such a procedure, a movable modal dialog will be created rather than the old modal dialog. You can do this from within PickColor because you're now able to pass update events to windows within the same application layer as the color picker.

PickColor also uses the new ColorSync color definition, so you can specify a color in any color space along with a destination profile. Likewise, a color can be returned in any color space.

PICKCOLOR PARAMETER BLOCK

Listing 1 shows the parameter block that you pass through to PickColor. The first two fields, theColor and dstProfile, are pretty obvious; they're simply the input (and output) color and the profile for the final output device. If there's no output device, you just set dstProfile to nil.

The flags field is a little more complicated. (It's also used in many of the low-level calls.) With PickColor, there are three flags you need to worry about:

- CanModifyPalette. If you set this flag, you're telling the color picker component that it's able to install a palette of its own that may modify (but not animate) the current color table. If you don't want the colors in your document to change as you make choices in the color picker dialog, don't set this flag.

Listing 1. PickColor parameter block

```
typedef struct ColorPickerInfo {
   PMColor              theColor;
   CMProfileHandle      dstProfile;
   long                 flags;
   DialogPlacementSpec  placeWhere;
   Point                dialogOrigin;
   long                 pickerType;
   UserEventProc        eventProc;
   ColorChangedProc     colorProc;
   long                 colorProcData;
   Str255               prompt;
   MenuItemInfo         mInfo;
   Boolean              newColorChosen;
} ColorPickerInfo;
```

- CanAnimatePalette. This flag is similar to CanModifyPalette, except that it allows the color picker component to modify or animate the palette as much as it wants to.
- AppIsColorSyncAware. This informs the Color Picker Manager that your application understands ColorSync colors. This means that a color may be returned to you in a different space than the one you passed in. For example, you could pass an RGB color (with no profile) to the Color Picker Manager and receive back a CMYK color (with its associated profile). If you don't set this flag, the Color Picker Manager automatically converts any color it receives back from the color picker component to RGB space.

The placeWhere field tells the Color Picker Manager where to position the color picker dialog. The choices are kAtSpecifiedOrigin (at the point specified by the dialogOrigin field), kDeepestColorScreen (centered on the deepest color screen), and kCenterOnMainScreen (centered on the main screen).

The dialogOrigin field (in conjunction with kAtSpecifiedOrigin) is used when you request that the color picker dialog be placed at a specific point. When PickColor returns, this field contains the location of the color picker dialog at the time it was closed.

You use the pickerType field to specify the component subtype of the color picker to select initially. If you set this field to 0, the default system color picker will be used (the last color picker chosen by the user). When PickColor returns, this field contains the component subtype of the color picker that was open when the user closed the dialog.

You should set the eventProc field to point to an event filter procedure that will handle events meant for your application. If this procedure returns true, the Color Picker Manager won't process the event further. If it returns false, the Color Picker Manager will handle the event if it was meant for the color picker. If you set this field to nil, a modal dialog will be created (rather than a movable modal dialog).

The colorProc field can contain a pointer to a procedure that will be called whenever the color changes. This allows live updating of colors in application documents as the

user selects them. The colorProcData field contains a long integer that's passed to the color-changed procedure and can be used for any private data.

The prompt field is a prompt string that the color picker displays to give the user some indication as to what the new color is for (for example, a highlight color).

The mInfo field tells the Color Picker Manager what the Edit menu ID is and where the various menu items are located within it.

The newColorChosen field is set on return from PickColor. If true, it means that the user chose a color and clicked OK; otherwise, the user clicked Cancel.

IMPLEMENTING PICKCOLOR

Now let's look at an example of how all this would be used. Listings 2 and 3 show two callbacks — the event filter procedure (MyEventProc) and the color-changed procedure (MyColorChangedProc). In the color-changed procedure we assume that ColorSync is installed. This is because we'll be setting the AppIsColorSyncAware flag when we call PickColor, so a non-RGB color might come back from the picker and, if so, you need to call ColorSync to convert it to RGB.

Once you have the two callback procedures, you can go ahead and call PickColor (see Listing 4).

USING THE LOW-LEVEL CALLS

The low-level Color Picker Manager calls are designed to allow tight integration of an application and a color picker (a floating palette, for example). Two features make this possible: the application can specify the type of dialog to put the color picker in, and the application maintains control over the event loop.

You can create three types of color picker dialogs with the low-level calls: system-owned, application-owned, and color picker–owned.

- A *system-owned* dialog is exactly like the dialog created by the high-level calls — it has OK, Cancel, and More Choices buttons. However, with the low-level calls, you can make the dialog modal, movable modal, or modeless.
- An *application-owned* dialog is actually owned (and supplied) by the application. You can use this type of dialog to integrate the color picker with other application window features or to extend the controls of the color picker. For example, you could add controls for altering the style of an object as well as its color.
- A *color picker–owned* dialog is created and owned by the color picker component itself, which gives that component great flexibility because it can determine the size and shape of the color picker and the dialog (color pickers in system-owned and application-owned dialogs are always the same size). This is useful for implementing floating pickers (such as the color wheel in Color MacCheese).

The application interacts with all three types of dialogs in the same way once they're created. The rest of this section describes how to create each type and then moves on to discuss how the application interacts with the color pickers, no matter what type of dialog you use. In other words, the type of dialog a color picker is in is abstracted enough that the application can use roughly the same code to handle all three types.

Listing 2. Event filter procedure

```
WindowPtr   myDocWindow;

pascal Boolean MyEventProc(EventRecord *event) {
   Boolean  handled = false;  // Assume we don't handle the event.

   switch (event->what) {
      case updateEvt:
         // Check to see if the update is for our window.
         if ((WindowPtr) event->message == myDocWindow) {
            DoTheUpdate(myDocWindow);
            handled = true;
         }
   }
   return handled;
}
```

Listing 3. Color-changed procedure

```
pascal void MyColorChangedProc(long userData, PMColorPtr newColor) {
   GrafPtr  port;
   CWorld   cWorld;
   CMColor  color;
   CMError  cwError;

   GetPort(&port);
   SetPort(myDocWindow);

   // Now check to see if the color has a profile. If so, we need to
   // convert it to RGB space.
   if (newColor->profile) {
      // Create a color world and convert the color. This color world
      // matches from the color's space to the system space (RGB).
      cwError = CWNewColorWorld(&cWorld, newColor->profile, 0L);
      if (cwError == noErr || cwError == CMProfilesIdentical) {
         // We created the color world. Now match the color using a copy
         // so that we don't munge the original.
         color = newColor->color;
         CWMatchColors(cWorld, &color, 1);
         CWDisposeColorWorld(cWorld);
      }
   } else
      color.rgb = newColor->color.rgb;

   // Set the new color and paint the port with it.
   myRGBColor = color.rgb;
   RGBForeColor(&color.rgb);
   PaintRect(&myDocWindow->portRect);

   SetPort(port);
}
```

Listing 4. Calling PickColor

```
ColorPickerInfo   cpInfo;
PMColor           savedColor;

// Set the input color to be an RGB color in system space.
cpInfo.theColor.color.rgb = myRGBColor;
cpInfo.theColor.profile = 0L;

cpInfo.dstProfile = 0L;
cpInfo.flags = AppIsColorSyncAware | CanModifyPalette | CanAnimatePalette;

// Center the picker on the deepest color screen.
cpInfo.placeWhere = kDeepestColorScreen;

// Use the default picker.
cpInfo.pickerType = 0L;

// Install the callbacks.
cpInfo.eventProc = MyEventProc;
cpInfo.colorProc = MyColorChangedProc;
cpInfo.colorProcData = 0L;

strcpy(cpInfo.prompt,"\pChoose a new color");

// Tell the Color Picker Manager about the Edit menu.
cpInfo.mInfo.editMenuID = kMyEditMenuID;
cpInfo.mInfo.cutItem = kMyCutItem;
cpInfo.mInfo.copyItem = kMyCopyItem;
cpInfo.mInfo.pasteItem = kMyPasteItem;
cpInfo.mInfo.clearItem = kMyClearItem;
cpInfo.mInfo.undoItem = kMyUndoItem;

// Save the current color, in case the user cancels.
savedColor = cpInfo.theColor;

// And finally, pick that color!
if (PickColor(&cpInfo) == noErr && cpInfo.newColorChosen)
   // Go use this new color. Remember it can be in any color space.
   DoNewColorStuff(&cpInfo.theColor);
else
   // Canceled or an error; restore old color.
   DoNewColorStuff(&savedColor);
}
```

CREATING THE DIALOG

When creating a system-owned dialog, the application needs to choose whether the dialog will be modal, movable modal, or modeless. This is handled by the use of two flags: DialogIsModal and DialogIsMoveable. Through obvious combinations of these flags, all three dialog types can be created. A nonmovable, modeless dialog (neither flag set) is illegal.

Listing 5 shows the code used to create a modeless system-owned dialog.

Listing 5. Creating a modeless system-owned dialog

```
SystemDialogInfo  sInfo;
OSErr             result;

sInfo.flags = DialogIsMoveable + AppIsColorSyncAware + CanModifyPalette
   + CanAnimatePalette;
sInfo.pickerType = 0L;
sInfo.placeWhere = kDeepestColorScreen;

sInfo.mInfo.editMenuID = kMyEditMenuID;
sInfo.mInfo.cutItem = kMyCutItem;
sInfo.mInfo.copyItem = kMyCopyItem;
sInfo.mInfo.pasteItem = kMyPasteItem;
sInfo.mInfo.clearItem = kMyClearItem;
sInfo.mInfo.undoItem = kMyUndoItem;

result = CreateColorDialog(&sInfo, &myPicker);
```

Listing 6 shows how to add a color picker to an application's own dialog (application-owned dialog).

Listing 6. Creating an application-owned dialog

```
DialogPtr              myDialog;
ApplicationDialogInfo  aInfo;
OSErr                  result;

// First create the dialog (make sure it's a color dialog so that the
// color picker can do all the color stuff it needs to do!).
myDialog = GetNewDialog(kMyDialogID, nil, (WindowPtr)-1);

// Set up the ApplicationDialogInfo structure.
aInfo.flags = DialogIsMoveable + AppIsColorSyncAware + CanModifyPalette
   + CanAnimatePalette;
aInfo.pickerType = 0L;
aInfo.theDialog = myDialog;

// Put the color picker's origin at (0,0) in the dialog.
aInfo.pickerOrigin.h = 0;
aInfo.pickerOrigin.v = 0;

// Set the Edit menu information.
aInfo.mInfo.editMenuID = kMyEditMenuID;
aInfo.mInfo.cutItem = kMyCutItem;
aInfo.mInfo.copyItem = kMyCopyItem;
aInfo.mInfo.pasteItem = kMyPasteItem;
aInfo.mInfo.clearItem = kMyClearItem;
aInfo.mInfo.undoItem = kMyUndoItem;

// Finally, add the color picker to the dialog.
result = AddPickerToDialog(&aInfo, &myPicker);
```

Listing 7. Creating a color picker–owned dialog

```
PickerDialogInfo  pInfo;
OSErr             result;

pInfo.flags = DialogIsMoveable + AppIsColorSyncAware + CanModifyPalette
   + CanAnimatePalette;
pInfo.pickerType = 0L;

pInfo.mInfo.editMenuID = kMyEditMenuID;
pInfo.mInfo.cutItem = kMyCutItem;
pInfo.mInfo.copyItem = kMyCopyItem;
pInfo.mInfo.pasteItem = kMyPasteItem;
pInfo.mInfo.clearItem = kMyClearItem;
pInfo.mInfo.undoItem = kMyUndoItem;

result = CreatePickerDialog(&pInfo, &myPicker);
```

Listing 7 shows how to create a color picker–owned dialog. As you can see, the code to create all three types of dialogs is nearly identical. Likewise the code to manage them after creation is very similar. Any explicit differences or requirements will be pointed out and explained as they're encountered.

SETTING AND GETTING THE CURRENT COLOR

One of the most obvious requirements for making a color picker useful is that there be a way to set and get the current color. This is very simple. Complexities arise only if you need to convert colors from the space they're returned in to a space you can understand (such as RGB). The following examples assume you're familiar enough with ColorSync to do this (Listing 2 shows how to convert from any space to system RGB space). If you don't want to deal with this, don't set the AppIsColorSyncAware flag and the Color Picker Manager will automatically convert any color it gets back from the color picker to RGB.

The concepts of original color and new color have been carried through from the old Color Picker Package to the new Color Picker Manager. Simply put, the *original color* is the color that the user is about to change and the *new color* is the color to which the user changes it. When setting the color for a color picker, you need to set both colors. Suppose, for example, that you're writing an object-based paint program and have created a floating color picker. When the user clicks an object, you want the color picker to show the color of that object. You would do this by setting the original color and new color for the color picker to the current color of that object. As the user changes the color of the object, the original color would remain the same and the new color would change. This gives feedback as to what would happen if the user were to cancel the color change. The code to do this is very simple:

```
void SetPickerToColor(RGBColor *rgb) {
   PMColor  aColor;

   aColor.color.rgb = *rgb;
   aColor.profile = 0L;
   SetPickerColor(myPicker, kOriginalColor, &aColor);
   SetPickerColor(myPicker, kNewColor, &aColor);
}
```

Whenever the user changes the current color, you need to be able to get the new color so that you can update your object accordingly:

```
void GetCurrentColor(RGBColor *rgb) {
   PMColor  aColor;

   GetPickerColor(myPicker, kNewColor, &aColor);
   *rgb = aColor.color.rgb;
}
```

Some of you might be saying, "But wait, this example is stupid. Isn't that what the color-changed callback is for?" The answer is yes, in the modal case, when the color-changed procedure is the only way the application knows that the color changed. In the modeless case, as we'll see below in the section "Giving Events to the Color Picker," the application is informed in other ways when the color changes. So in the modeless case, you might want to view the colors that the color-changed procedure provides you with as temporary colors and not update your internal data until the user has actually chosen a color (or at least stopped dragging on a slider). You should then make an explicit call to the Color Picker Manager to get the color, and update your internal data.

SETTING THE DESTINATION PROFILE

If you're picking a color for an output device for which you have a ColorSync profile, you can give this profile to the color picker component so that it can communicate the profile's information to the user (assuming it knows how). You do this with a simple call, SetPickerProfile. There's also a matching call, GetPickerProfile, to get the current profile from the color picker. It's important to remember that the application owns the memory of any profiles it gives or receives from the color picker. When you set the destination profile, the color picker component makes a copy of the profile handle; when you get the destination profile, you give the color picker component a handle into which it copies the profile data. The following code shows how to set and get the destination profile. Setting it is optional; the color picker assumes that there's no profile unless you explicitly set one.

```
void SetDestinationProfile(CMProfileHandle profile) {
   if (SetPickerProfile(myPicker, profile) != noErr)
      HandleError();
}

void GetDestinationProfile(CMProfileHandle profile) {
   if (GetPickerProfile(myPicker, profile) != noErr)
      HandleError();
}
```

GIVING EVENTS TO THE COLOR PICKER

The basic model for giving events to the color picker is similar to DialogSelect. For the most part, you give the event to the Color Picker Manager through the DoPickerEvent call. It either handles the event or returns it to the application for the application to handle.

There's one exception to this rule: menus. If you've created a modal system dialog, the Color Picker Manager can handle the Edit menu events for you (as it does when you call PickColor). However, for modeless color pickers there are many menu items that the Color Picker Manager has no idea how to handle. If you do send these events through to the Color Picker Manager, it will assume all Edit menu selections are

meant for the color picker and ignore everything else. Therefore, with modeless dialogs, the application needs to be sure to handle its own menu events before calling DoPickerEvent.

You'll also need to do some extra work in order for the Color Picker Manager to handle the Edit menu correctly. If an Edit menu choice will be for the color picker (that is, the color picker dialog is frontmost and the current text item in the dialog belongs to the color picker), you need to set up the Edit menu as the Color Picker Manager and color picker component want it. To determine how they want the Edit menu, call GetPickerEditMenuState. If the user does choose an Edit menu item, the application needs to call DoPickerEdit to tell the Color Picker Manager which edit operation to perform. There's more on this later under "Handling the Edit Menu."

Each time you call DoPickerEvent and the color picker component or the Color Picker Manager handles the event, it returns a constant describing what happened. There are several possible results, which are listed in Table 1.

Table 1. DoPickerEvent return constants

Constant	Meaning
kDidNothing	Nothing happened that's worth reporting.
kColorChanged	The user changed the color; you may need to call GetPickerColor to get the new color.
kOKHit	The user clicked OK; returned only by system- or color picker–owned dialogs.
kCancelHit	The user clicked Cancel; returned only by system- or color picker–owned dialogs.
kNewPickerChosen	The user chose a new color picker from the More Choices list; returned only by system-owned dialogs.
kApplItemHit	The Dialog Manager returned an item intended for one of the application's dialog items; returned only by application-owned dialogs.

Internally, the Color Picker Manager handles the event by calling DialogSelect and then processing the event from there. If the color picker is in an application dialog and an application item is selected, the Color Picker Manager returns kApplItemHit as well as the item number.

There are a few things to keep in mind regarding the DoPickerEvent return constants. How you handle kColorChanged with application dialogs depends on your application; for system-owned and color picker–owned dialogs you probably should wait until the user clicks OK before treating the color as final. With kOKHit, you should save the new color and close the dialog. With kCancelHit, you should restore the old color and dispose of the color picker. If kApplItemHit is returned, you need to handle the event as you would for the Dialog Manager. You don't need to care about kNewPickerChosen, which happens only with a system-owned dialog.

If you have a color-changed procedure for the color picker to call, you supply the procedure, along with any data it needs to be called with, to DoPickerEvent.

Listing 8 shows what your event loop might look like. In this code we assume that you always want to handle the menu events yourself, as discussed above.

Listing 8. Sample event loop

```
#define IsMenuKey(x) ((x)->what == keyDown && (x)->modifiers & cmdKey)
Boolean SampleDoEvent(EventRecord *event) {
   Boolean     handled = false, isMenuEvent = false;
   EventData   pEvent;
   short       inWhere;
   WindowPtr   whichWindow;

   // We are assuming that the application always wants to handle menus.
   if (event->what == mouseDown) {
      inWhere = FindWindow(event->where, &whichWindow);
      if (inWhere == inMenuBar)
         isMenuEvent = true;
   }
   if (isMenuEvent || IsMenuKey(event)) {
      DoMenu(event);
      handled = true;
   }

   // If the event's not handled yet, call the Color Picker Manager to
   // give it a shot.
   if (!handled) {
      pEvent.event = event;
      pEvent.colorProc = MyModelessColorChangedProc;
      pEvent.colorProcData = 0L;
      DoPickerEvent(myPicker, &pEvent);
      handled = pEvent.handled;

      // If the color picker handled it, we might want to do something
      // with the results.
      if (handled) {
         switch (pEvent.action) {
            case kDidNothing:
               break;
            case kColorChanged:
               UseNewColor(myPicker);
               break;
            case kOKHit:
               UseNewColor(myPicker);
               DisposeColorPicker(myPicker);
               myPicker = nil;
               break;
            case kCancelHit:
               UseOriginalColor(myPicker);
               DisposeColorPicker(myPicker);
               myPicker = nil;
               break;
            case kNewPickerChosen:
               // You shouldn't care about this.
               break;
```

(continued on next page)

Listing 8. Sample event loop *(continued)*

```
            case kApplItemHit:
               // Handle the item as you would for the Dialog Manager.
               HandleAppItem(pEvent.itemHit);
               break;
         }
      }
   }
   if (!handled) {
      // The event hasn't been handled. Treat it as you would any normal
      // Macintosh event. If you have other dialogs, you need to call
      // DialogSelect. Remember, if the event is a mouseDown, you
      // already called FindWindow!
   }
   return handled;
}
```

FORECAST EVENTS

When dealing with a color picker, you'll sometimes need to warn it about a user action that might affect it. For example, if you have a color picker in an application dialog and the user closes that dialog, you might want to see if the color picker is in a state that can handle this. If the user had just typed some numbers into the color picker that left it in an inconsistent state, it would be nice if the color picker could have a chance to complain to the user before it was indiscriminately closed.

You can do this by using *forecast events*. These aren't really events in themselves, but are warnings to the color picker. To send forecast events to the color picker component, you use the same call as for regular events — DoPickerEvent — except that you set the event field to nil and set the forecast field to an appropriate constant. The color picker component tells you whether it's ready for the action to occur by setting the handled field of the EventData structure to true if it's not ready and false if it is.

For the most part, the only time your application needs to worry about this is when the color picker is about to be closed. If the Color Picker Manager has instigated the closing (such as when the action field is set to kOKHit after you called DoPickerEvent), you don't need to worry about telling the color picker component because the Color Picker Manager has already done so. However, if the user has just clicked the window's close box (for an application dialog) or has chosen Close from a menu, you should send a forecast event to the color picker component.

The following example shows a function called CheckIfPickerCanClose. If this function returns true, the color picker can close; otherwise, it can't close for some reason. It's safe to assume that the color picker has informed the user of the problem.

```
Boolean CheckIfPickerCanClose() {
   EventData   pEvent;

   pEvent.event = 0L;      // Make it a forecast event.
   pEvent.forcast = kDialogAccept;
   DoPickerEvent(myPicker, &pEvent);

   return !pEvent.handled;
}
```

HANDLING THE EDIT MENU

As mentioned earlier, the Edit menu takes some special work. In addition to standard menu processing, if an Edit menu choice will be for the color picker, you need to set the state of the Edit menu items according to the color picker specifications and, if an Edit menu item is chosen, send the appropriate message to the color picker. This is done through two simple calls: GetPickerEditMenuState and DoPickerEdit.

Once you've determined that there has been a mouse-down event in the menu bar or a keyboard equivalent has been pressed, you need to determine who owns the Edit menu. If the color picker is in a color picker–owned or system-owned dialog and it's frontmost, the color picker obviously owns it. If the color picker is in an application-owned dialog and it's frontmost, ownership of the Edit menu depends on the current item. The choice really depends on your application. As a general rule, whoever owns the current item owns the Edit menu. If you do call DoPickerEdit while the current item belongs to your application, it will implement the standard cut, copy, paste, and clear features for you. If your application needs to do more than this, you'll need to handle it yourself.

In Listing 9 we assume that the owner of the current item owns the Edit menu. The item number for the application's last dialog item is kMyLastItem. If you have a system-owned or color picker–owned dialog, this constant should be set to 0. In an application-owned dialog the picker's items will always be added after the application's, so your item numbers remain the same.

Listing 9. Handling the Edit menu

```
Boolean DoMenu(EventRecord *event) {
   long          mChoice;
   EditData      eData;
   EditOperation eOperation;

   // If picker is in front and current edit item is the picker's,
   // set up the Edit menu the way the picker wants it.
   if (FrontWindow() == myDialog &&
         ((DialogPeek)myDialog)->editField + 1 > kMyLastItem) {
      MenuState     mState;
      MenuHandle    theMenu;

      GetPickerEditMenuState(myPicker, &mState);
      theMenu = GetMenu(kMyEditMenuID);
      if (mState.cutEnabled)
         EnableItem(theMenu, kMyCutItem);
      else
         DisableItem(theMenu, kMyCutItem);
      if (mState.copyEnabled)
         EnableItem(theMenu, kMyCopyItem);
      else
         DisableItem(theMenu, kMyCopyItem);
      if (mState.pasteEnabled)
         EnableItem(theMenu, kMyPasteItem);
      else
         DisableItem(theMenu, kMyPasteItem);
```

(continued on next page)

Listing 9. Handling the Edit menu *(continued)*

```
        if (mState.clearEnabled)
            EnableItem(theMenu, kMyClearItem);
        else
            DisableItem(theMenu, kMyClearItem);
        if (mState.undoEnabled) {
            SetItem(theMenu, kMyUndoItem, mState.undoString);
            EnableItem(theMenu, kMyUndoItem);
        }
        else
            DisableItem(theMenu, kMyUndoItem);
    }

    // Give the event to the Menu Manager.
    if (event->what == mouseDown)
        mChoice = MenuSelect(event->where);
    else
        mChoice = MenuKey(event->message);

    // If not the Edit menu, handle normally.
    if (HiWord(mChoice) != kMyEditMenuID) {
        HandleMenuChoice(mChoice);
        return true;
    }

    switch (LoWord(mChoice)) {
        case kMyCutItem:
            eOperation = kCut;
            break;
        case kMyCopyItem:
            eOperation = kCopy;
            break;
        case kMyPasteItem:
            eOperation = kPaste;
            break;
        case kMyClearItem:
            eOperation = kClear;
            break;
        case kMyUndoItem:
            eOperation = kUndo;
            break;
        default:
            eOperation = -1;
            break;
    }
    if (eOperation >= 0) {
        eData.theEdit = eOperation;
        DoPickerEdit(myPicker, &eData);
        // Ignore the results here; assume that the color changed.
        UseNewColor(myPicker);
    }
    HiliteMenu(0);
    return true;
}
```

USING BALLOON HELP

The Color Picker Manager provides support for Balloon Help. Most applications don't need to do anything special for Balloon Help to work for a color picker in any type of dialog. However, for applications in which you need more control over Balloon Help, you can call ExtractPickerHelpItem to get the balloon for the color picker. It's up to the application to determine whether the cursor is over a color picker's item or one of its own. The best way to do this is to check to see if it's over one of the application items. If so, put up your own balloon; otherwise, call ExtractPickerHelpItem and put up the balloon it returns. ExtractPickerHelpItem will ask the color picker for a balloon and search the color picker's help resource for an appropriate balloon. If it can't find one, it returns the error noHelpForItem.

The hardest part about using ExtractPickerHelpItem is determining which item the cursor is over. Fortunately, there's a Dialog Manager call, FindDItem, that does the dirty work for you. Listing 10 shows how you would use these calls. Everything in this example is actually done by the Color Picker Manager internally; the example just gives you a general idea of how to use the ExtractPickerHelpItem call.

Listing 10. Using ExtractPickerHelpItem

```
void DoBalloonHelp(void) {
   HelpItemInfo   helpInfo;
   short          itemNo;
   Point          where;
   OSErr          err;

   GetMouse(&where);
   itemNo = FindDItem(myDialog, where) + 1;

   // Go and get the color picker's help item.
   helpInfo.options = 0;
   helpInfo.tip.v = helpInfo.tip.h = 0;
   SetRect(&helpInfo.altRect, 0, 0, 0, 0);
   helpInfo.theProc = 0;
   helpInfo.variant = 0;
   helpInfo.helpMessage.hmmHelpType = 0;
   helpInfo.helpMessage.u.hmmPictHandle = 0L;
   err = ExtractPickerHelpItem(myPicker, itemNo, 0, &helpInfo);

   // Show the balloon if we found one.
   if (err == noErr) {
      // If altRect is empty, we need to use the item's rectangle.
      if (EmptyRect(&helpInfo.altRect)) {
         short    iType;
         Handle   iHandle;

         GetDItem(myDialog, itemNo, &iType, &iHandle, &helpInfo.altRect);
      }

      // Convert the tip to dialog coordinates.
      helpInfo.tip.h += helpInfo.altRect.left;
      helpInfo.tip.v += helpInfo.altRect.top;
```

(continued on next page)

Listing 10. Using ExtractPickerHelpItem *(continued)*

```
        // Convert the tip and altRect to global coordinates.
        LocalToGlobal(&helpInfo.tip);
        LocalToGlobal((Point *) &helpInfo.altRect.top);
        LocalToGlobal((Point *) &helpInfo.altRect.bottom);

        // Finally, put the balloon up.
        HMShowBalloon(&helpInfo.helpMessage, helpInfo.tip,
           &helpInfo.altRect, 0L, helpInfo.theProc, helpInfo.variant,
           kHMRegularWindow);
    }
}
```

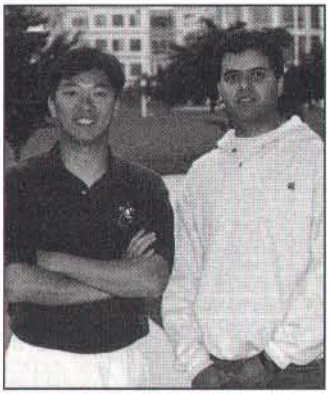
TAKE YOUR PICK

You should now have a general idea of how to use the new Color Picker Manager. Most applications will need only the high-level calls. However, developers who use color more thoroughly may want to take advantage of the low-level calls. The low-level calls were designed to be very flexible and easy to use. The simple implementations shown in this article are trivial; more complicated uses are possible, and shouldn't be much harder to write.

Having experimented with the new features of Color Picker 2.0, you may still want to write your own color picker component — for example, to implement your own floating color picker. The new Color Picker Manager makes it easier for you to write your own color picker component and allows you to share it among several applications (and make it available for general system use as well).

So take your pick of the color pickers already available through the high-level or low-level calls or move beyond this article to create your own custom color picker component. Either way, you're looking at a colorful future with Color Picker 2.0.

Thanks to our technical reviewers Don Moccia, Konstantin Othmer, and David Surovell. •



SOMEWHERE IN QUICKTIME

Media Capture Using the Sequence Grabber

JOHN WANG AND FERNANDO URBINA

A very important and often overlooked feature of QuickTime is the standardization of media capture. Since its initial release, QuickTime has defined an API for capturing different types of media, including video and sound. This API, known as the sequence grabber, makes it possible to easily add media capture to any application.

Not only are applications that use the sequence grabber able to automatically support any QuickTime-compatible media capture hardware, but they also perform flawlessly and efficiently regardless of system configuration. This is not an easy task considering all the variations in hardware features and system configurations. In fact, we're even hesitant to say that the sequence grabber "supports video and sound capture" because the sequence grabber API also insulates the programmer from the actual media type being captured. The sequence grabber supports any media type, and, with the release of QuickTime 2.0, users can automatically capture the new music media type in addition to sound and video.

To demonstrate the proper use of the sequence grabber, we've included on this issue's CD a simple, but complete, sequence grabber application — all in about 10K of compiled C code! If you're looking for a general all-purpose capture application that's efficient, reliable, and best of all, customizable, look no further. After a brief introduction to the sequence grabber, we'll discuss the sample code, and then end with some special considerations for media capture on Macintosh AV models.

WHAT IS THE SEQUENCE GRABBER?

The sequence grabber is actually a component of type 'barg' (read it backwards). Although the specification for the component is completely defined in Chapter 5 of *Inside Macintosh: QuickTime Components*, it's very unlikely that you'll ever want to implement your own 'barg' component. Instead, you'll be using this component specification as the API definition for the standard sequence grabber.

The sequence grabber component implements the basic functionality of media capture. For handling specific media-related functions, the sequence grabber calls on various sequence grabber channel components (as defined in Chapter 6 of *Inside Macintosh: QuickTime Components*); there's one such component for each media type. Before QuickTime 2.0, the two standard channel components available were the video and sound sequence grabber channels, enabling the sequence grabber to capture video and sound media. QuickTime 2.0 includes the new music sequence grabber channel, allowing real-time capture of music from MIDI instruments.

Sequence grabber panel components (described in Chapter 7 of *Inside Macintosh: QuickTime Components*) manage items in a settings dialog box that allows the sequence grabber to obtain configuration information from a user. Applications typically don't use sequence grabber panel components directly; instead, the sequence grabber automatically uses them for relevant sequence grabber component calls.

USING THE SEQUENCE GRABBER

Using the sequence grabber is as simple as opening the sequence grabber component and calling SGInitialize (complete error checking can be found in the sample code on the CD):

```
theSG =
  OpenDefaultComponent(SeqGrabComponentType, 0);
SGInitialize(theSG);
```

JOHN WANG (AppleLink WANG.JY) While writing the sequence grabber sample code for this column, John watched the movie *Top Gun* so many times that he can now duplicate each and every air combat scene with his favorite flight simulator, FA/18 Hornet. John once aspired to become a private pilot, but that idea was quickly quelled once his significant others found out. As Skate so succinctly put it, "Woof woof wooof?" Translation: "Who's going to feed me if you kill yourself?"•

FERNANDO ("NANO") URBINA (AppleLink NANO) uses his Macintosh AV to capture the views of the Rockies from his home office in Colorado Springs. He still doesn't understand how it can thunder and snow at the same time, but thinks he'll be able to figure this out once he adjusts to the lack of oxygen. Nano suffers severe withdrawal from his favorite coffee shop near the Apple campus in Cupertino, but manages to get a fix about once a month when he returns there. He worked on the original AV models and is now a member of the second-generation AV team.•

It's also important to call SGSetGWorld to set the window used for displaying any visual data.

```
SGSetGWorld((**myWindowInfo).theSG,
   (CGrafPtr) myWindow, nil);
```

Opening the channel components. Now it's a matter of calling SGNewChannel to open a sequence grabber channel component to access a particular channel media type. However, rather than hard-coding the media types into the sample application, as in the call

```
SGNewChannel(theSG, VideoMediaType,
   &videoChannel);
```

it's better to use the Component Manager to search for all the different sequence grabber channel components and open a connection to each one. This guarantees that the capture application can automatically support new media types such as the music media type in QuickTime 2.0.

For example, the following code compiles a list of sequence grabber channel components:

```
cd.componentType = SeqGrabChannelType;
cd.componentSubType = 0;
cd.componentManufacturer = 0;
cd.componentFlags = 0;
cd.componentFlagsMask = 0;
aComponent = 0;

for (i=0, done=false; i<kMAXCHANNELS && !done;
      i++) {
   aComp = FindNextComponent(aComp, &cd);
   if (aComp != 0) {
      // Get the channel name and type.
      gSGInfo.channelName[i] = NewHandle(4);
      GetComponentInfo(aComp, &theCD,
         gSGInfo.channelName[i], nil, nil);
      gSGInfo.channelType[i] =
         theCD.componentSubType;
   } else
      done = true;
}
```

This list of component types can then be used to open a connection to each of the media types with SGNewChannel, or SGNewChannelFromComponent if the channel component is already open.

Saving and restoring settings. We want the sample application to start up each time with the same channel settings and video compression settings as when the application was last used. To implement this, we use a preferences file to store these settings. The compression settings are restored with two sequence grabber calls:

```
SGSetVideoCompressorType(
   (**myWindowInfo).channel[videoChannel],
   gSGInfo.cInfo.compressorType);

SGSetVideoCompressor(
   (**myWindowInfo).channel[videoChannel],
   gSGInfo.cInfo.depth, nil,
   gSGInfo.cInfo.spatialQuality,
   gSGInfo.cInfo.temporalQuality,
   gSGInfo.cInfo.keyFrameRate);
```

The channel settings are restored by a simple call to SGSetChannelSettings with the settings retrieved from the preferences file:

```
SGSetChannelSettings(theSG, channel[i],
   channelSettings[i], 0);
```

Previewing. We're almost ready to begin previewing. But note that some sequence grabber channel components require additional calls before they can be used. For instance, spatial channels such as video require a call to SGSetChannelBounds to set the channel's display boundary rectangle. So, once the channels are created and the previous settings are restored as discussed above, we make a call to SGSetChannelBounds for the video media to set the video capture to encompass the entire window. We also call SGSetChannelUsage for all sequence grabber channels, which tells the sequence grabber that we want to preview and record every channel.

To start previewing, we simply call SGStartPreview. However, while we're previewing, any changes to the system must be handled with care. First, we'll pause the preview whenever an event that requires updating of the channel information occurs. For example, if the capture window is dragged, we'll pause the video, move the window, and then unpause the video. Likewise, if we resize the window, we'll want to pause the preview, resize the window, and then unpause the preview:

```
// Pause the sequence grabber before resizing.
SGPause((**myWindowInfo).theSG, true);

// Resize and then update the video channel.
SizeWindow(theWindow, width, height, false);
MyUpdateChannels(theWindow);

// OK. We can restart again.
SGPause((**myWindowInfo).theSG, false);
```

Notice the call to MyUpdateChannels. This is a routine in the sample application that updates the video bounds and channel usage by calling SGSetChannelBounds and SGSetChannelUsage.

The user configuration dialog. Another feature that needs to be handled in a capture application is the user configuration dialog for each of the different capture medias. This is actually one of the simplest things to deal with because the sequence grabber component handles everything. It even stores the settings internally. To retrieve the settings, we can simply call SGGetChannelSettings at a later time. In the sample application, we get the channel settings before we close the connection to the sequence grabber. Then we save the settings in the preferences file.

This is all the code necessary to display and handle the user configuration dialog:

```
SGSettingsDialog(theSG, theChannel, 0, nil, 0,
   nil, 0);
```

Recording. The last important part of the sample code is sequence grabber recording. Before recording can begin, we need to specify an output file with SGSetDataOutput so that the sequence grabber knows where to save the captured media data:

```
StandardPutFile("\PName of new movie:", "\pMovie",
      &reply);
if (!reply.sfGood)
   return;
SGSetDataOutput(mySG, &reply.sfFile,
      seqGrabToDisk);
```

Then we start recording by simply calling

```
SGStartRecord(theSG);
```

We loop and call SGIdle until the mouse button is pressed to stop recording. This is the most efficient way to record: we don't want to call WaitNextEvent, since that would give other processes time. Instead, we want to hog the CPU time until the recording process is done.

```
while (!Button() && !err) {
   err = SGIdle(theSG);
}
```

We stop recording and start previewing again as follows:

```
SGStop(mySG);
SGStartPreview((**myWindowInfo).theSG);
```

And, of course, just to be nice, we flush the mouse-down events so that no application switching takes place after the mouse button is pressed:

```
FlushEvents(mDownMask, 0);
```

That's really all there is to the sequence grabber sample application.

SPECIAL CONSIDERATIONS FOR AV MODELS

As mentioned earlier, one of the key features of the sequence grabber is its ability to work with all hardware and system configurations. This is not an easy task considering all the different types of video capture boards. For example, there are boards that are simply frame grabbers, and there are those that support on-board hardware video compression. To make every configuration work, the sequence grabber has to handle every case. Here we'll discuss the unique features of the Macintosh AV models and some steps you can take to improve their capture rate.

The video-in circuitry allows the AV models to display 16-bit color and 8-bit grayscale. And, although the hardware can't display video-in at 24 bits per pixel, you can capture video using YUV 4:2:2 compression and achieve an effective 24 bits per pixel. To capture in YUV, you must use the AV's video digitizer hardware compression feature, which you can do simply by selecting "Component Video - YUV" from the list of compressors in the Compression panel of the video settings dialog. You should also make sure that you haven't checked a "Post Compress Video" or similar checkbox in a movie-grabbing application. Selecting this checkbox would bypass the hardware compression, and the sequence grabber would grab the data in raw RGB format.

The AV circuitry can't display video when it's capturing the compressed data. The sequence grabber realizes that it needs to decompress the data into the capture window in order to give the visual feedback that's normally expected. This is fine and dandy, but since there's no hardware decompression in the system, the image decompression is completed in software. This degrades the capture rate.

Knowing that the decompression during recording is what's hurting the capture rate, you can easily rectify the problem by turning off preview during recording so that decompression into the capture window won't take place. To do this, you just call SGSetChannelUsage for the video channel with the seqGrabPlayDuringRecord flag set to 0. In the sample code, a menu selection allows you to turn off video playthrough during recording.

The downside of using YUV compression is that playback without hardware decompression isn't very smooth because of the high data rate and raw processing power needed to decompress each pixel. After capturing, you should recompress the movie using a compressor such as Cinepak or Video that provides a better playback rate.

GO MAKE A MOVIE

The sequence grabber obviously makes the job of media capture simpler. But there are many other factors that can play a part. Hard drive transfer rate, disk fragmentation, SCSI bandwith, sound settings, and AppleTalk activity all play an important part in limiting the maximum capture rate. You can also maximize the capture rate by rebooting with no AppleTalk connections. You should also experiment with the different sound sample rates, as these also affect the capture rate.

New additions to the sequence grabber in QuickTime 2.0 also help. Instead of capturing to a single movie file, it's now possible to specify a different file for each channel. For example, you can record video to a large and fast external hard drive and record audio to the internal hard drive. This optimization allows for better allocation of resources and better efficiency because each channel has higher bandwidth. Using the sample code, if QuickTime 2.0 is installed, you can select recording to separate files.

There are, of course, other optimizations that can be explored. With a bit of creativity and testing, you *can* achieve the optimal capture rates.

RELATED READING

- *Inside Macintosh: QuickTime Components*, Chapters 5–7, and *Inside Macintosh: More Macintosh Toolbox*, Chapter 6, "Component Manager" (Addison-Wesley, 1993).
- "Video Digitizing Under QuickTime" by Casey King and Gary Woodcock, *develop* Issue 14. About the sequence grabber and video capture.

Thanks to Peter Hoddie and Don Johnson for reviewing this column.

Implementing Inheritance In Scripts

"Programming for Flexibility: The Open Scripting Architecture" in develop *Issue 18 showed you how to use scripts to increase your program's flexibility. This article builds on that one and explains how to implement an inheritance scheme in your application that will enable your AppleScript scripts to share handlers and properties and to support global variables. You'll also learn a way to support inheritance in other OSA languages with just a little extra work.*

PAUL G. SMITH

In Issue 18 of *develop*, I showed you how to attach scripts to application-domain objects and how to delegate the handling of Apple events to those scripts. I left you with a challenge: to figure out how to support global variables and to enable scripts to share subroutines and handlers. To meet this challenge you need to implement inheritance. The AppleScript 1.1 API gives you all the necessary calls to implement inheritance in embedded AppleScript scripts, but not all are documented yet in *Inside Macintosh*. This article documents the calls you need and describes an inheritance scheme that relies on them.

In a nutshell, here's the scheme:

1. Decide what kind of script inheritance hierarchy to use.
2. Link your scripts together in an inheritance chain.
3. Create a shared handlers script to define subroutines and Apple event handlers that are shared among all scripts. Make this script the parent of all other scripts in your program, in effect putting it at the end of the inheritance chain.
4. Create a global variables script and add this to the start of the inheritance chain so that it's the first script to receive incoming messages. Save this script to disk when the program exits and reload it when the program restarts, so that variables persist.

You can use much the same scheme to implement inheritance in other Open Scripting Architecture (OSA) languages, but more work is required to link scripts together in an inheritance chain, and you must forgo the luxury of sharing global variables between scripts. At the end of this article, the section "Inheritance in Other OSA Languages" describes the extra work your program must do.

PAUL G. SMITH (AppleLink COMMSTALK.HQ) took time out from his preferred occupation of snoozing on a beach to write this article. He also occasionally takes time out to write software for Full Moon Software Inc., provide consultancy services to corporate clients, and watch his cat, Mack, dismember his Macintosh's mouse. He was the lead developer of AgentBuilder and wrote ScriptWizard, the AppleScript debugger, before he found his true, totally prone, calling.•

The sample program SimpliFace2 on this issue's CD demonstrates the inheritance mechanisms discussed here. SimpliFace2 is an extension of SimpliFace, the basic interface builder used to illustrate the article in Issue 18. The SimpliFace2 sample code has a compile-time flag qUseOSAinheritance, defined in the header file SimpliFace2Common.h. If this flag is undefined, the program uses the AppleScript-specific inheritance scheme described in the bulk of this article. If the flag is defined, SimpliFace2 uses a general-purpose scheme that involves the extra work outlined in the section on inheritance in other OSA languages.

CHOOSE A SCRIPT INHERITANCE HIERARCHY

The first thing to do is to decide what kind of script inheritance hierarchy to use. You can use a runtime containment hierarchy (like that used by HyperCard and FaceSpan™), a class hierarchy (like that used by AgentBuilder), or some hybrid of the two, as demonstrated by SimpliFace2. Let's look at each of these hierarchy types in turn.

FaceSpan (formerly Frontmost) is the interface builder bundled with the AppleScript 1.1 Software Development Toolkit. Perhaps the best known OSA "client," FaceSpan was developed by Lee Buck (of "WindowScript" fame) of Software Designs Unlimited, Inc. AgentBuilder, from commstalk hq and Full Moon Software Inc., is a framework for the creation of communications and information-processing agents that uses embedded OSA scripts to customize agent behavior.•

Figure 1 shows a runtime containment hierarchy. In this kind of hierarchy, objects inherit behavior from their containers at run time. In the object containment hierarchy used by HyperCard, for example, the scripts of buttons and fields within cards are at the bottom of the hierarchy. Above them are the scripts of the cards that contain the buttons and fields, and above each card script is the script of the background that contains the card. Above each background script, in turn, is the script of the stack that contains all the backgrounds. The handlers in each container's script are shared by the scripts of all the objects it contains.

Figure 2 shows a class hierarchy. In this kind of hierarchy, objects inherit behavior from their parent classes. For instance, the behavior of AgentBuilder objects is defined in an "ancestor" object of each class, which is the parent of all instances of that class. This permits the standard scripted behavior of object classes to be overridden in derived class instances.

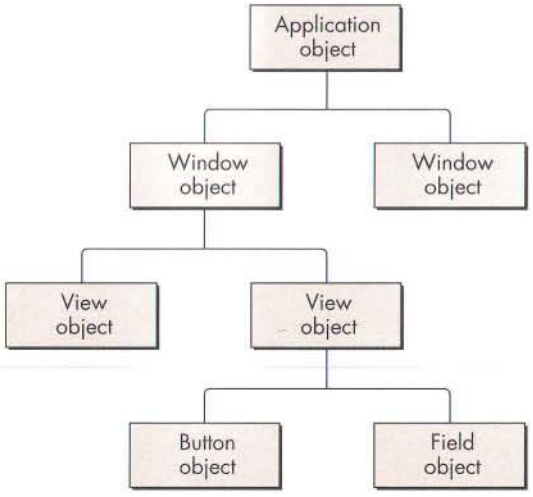


Figure 1. A runtime containment hierarchy

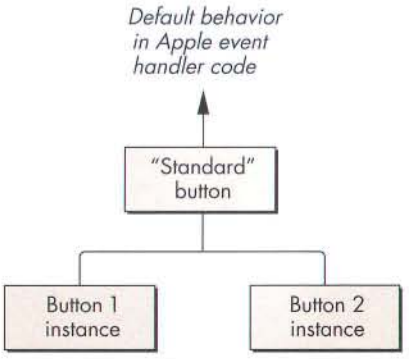


Figure 2. A class hierarchy

Figure 3 shows the hybrid script inheritance hierarchy used in SimpliFace2. In SimpliFace2, the scripts of user-interface objects — such as windows, labels, and buttons — are organized so that they inherit behavior from the runtime containment hierarchy. However, the script of the application object isn't included in the inheritance chain for the script of any user-interface object, and the shared handlers script becomes the ultimate parent of all other scripts. I chose to use this hybrid hierarchy in order to demonstrate a wider range of techniques, not for any reason intrinsic to the program design.

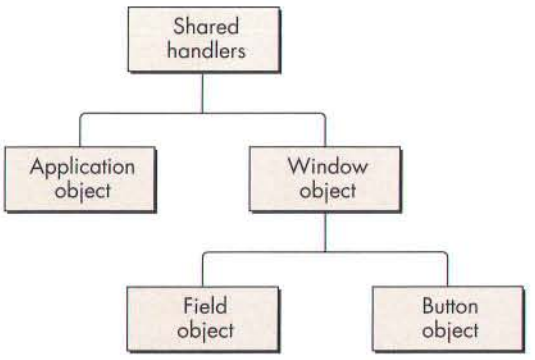


Figure 3. The hybrid script inheritance hierarchy used in SimpliFace2

The kind of script inheritance hierarchy to use depends on the nature of the messages being handled in your program. Using a class hierarchy is most appropriate if the messages are Apple events defined in the program's 'aete' resource. If the incoming messages are primarily user-defined subroutines being handled inside scripts, using a runtime containment hierarchy is probably more natural for the scripter.

Another way to look at this choice is that if you want to enable users to customize your program's capabilities by attaching scripts to application-domain objects, using a runtime containment hierarchy isn't always the best idea. Because different application-domain objects handle the same Apple event message in different ways (in other words, the semantic meaning of the message differs depending on what object it's directed at), unwanted side effects could result from an object's handling an Apple event message intended for a different level in the containment hierarchy. Using a class hierarchy ensures that messages will be dealt with only by objects of the class that understands them.

Once you've chosen the type of script inheritance hierarchy most appropriate for your program, you can link scripts together in an inheritance chain.

LINK SCRIPTS IN AN INHERITANCE CHAIN

Linking scripts together in an AppleScript inheritance chain is as simple as setting their parent properties. Before I tell you how to do that, though, let's review a few facts about script objects and inheritance. As mentioned in the Issue 18 article, a script context (a script compiled using the AppleScript OSA component) is equivalent to a script object in the AppleScript language, so everything I say here about script objects applies to script contexts as well.

ABOUT APPLESCRIPT SCRIPT OBJECTS AND INHERITANCE CHAINS

Script objects can contain global variables, properties, and handlers for Apple event messages and subroutine calls. A script object can have as its parent property an object specifier or another script object. Thus, one script object can become the

parent of another, and the child script object can inherit properties and handlers from the parent script object. Parent and child script objects are linked together in an inheritance chain; this is the path from child to parent to grandparent and so on in an inheritance hierarchy, as illustrated in Figure 4.

For (a lot) more on object specifiers, see "Apple Event Objects and You" by Richard Clark in *develop* Issue 10.•

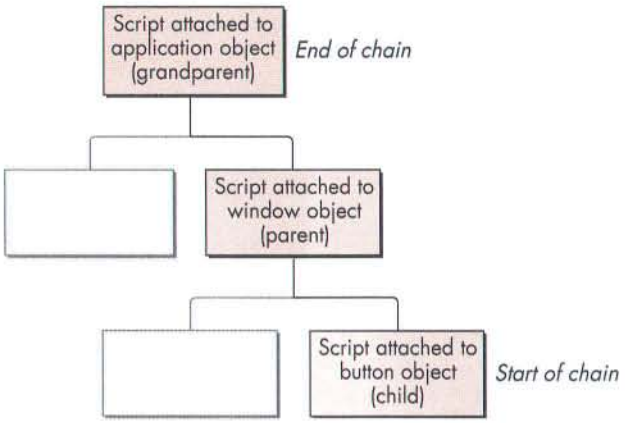


Figure 4. A script inheritance chain

An incoming Apple event message is received by the child script object at the start of the inheritance chain. If AppleScript can't resolve a reference to a handler or variable name within this script object, it searches through the entire inheritance chain to find it. The handler or variable is resolved wherever it's found in the inheritance chain. When a handler continues a message (that is, passes the message to its parent), AppleScript starts searching in its parent script object. Messages that target objects outside the program's domain, or that aren't handled anywhere in the script inheritance chain (such as Apple events defined in the program's 'aete' resource, which are handled in the program code instead), or that are continued out of the inheritance chain, are redispatched as Apple events.

SETTING A SCRIPT'S PARENT PROPERTY

Now that you understand the dynamics of script inheritance, I'll show you how to set a script's parent property and thus link it to an inheritance chain. In the AppleScript language, you simply say what you'd like the parent set to, as illustrated here:

```
script mom
   on getName()
      return "Fenella"
   end getName
end script

script toddler
   property parent : mom
   on getName()
      set myMom to continue getName()
      return "Bart, son of " & myMom
   end getName
end script

getName() of toddler --> returns "Bart, son of Fenella"
```

To set the parent of a script context from a programming language, you can use the AppleScript routine OSASetProperty. This general-purpose routine (defined in the header file ASDebugging.h, which was added with the AppleScript 1.1 API) accesses either a predefined property or a user-defined variable, depending on the AEDesc passed to it. To access a predefined property — the parent property — you create a descriptor of typeProperty (*not* typeType), specifying the property ID as the data. The parameters to the call are (1) the scripting component (probably the AppleScript component), (2) a mode flag (we use the null mode, indicating no special action should be taken; alternatively, we could instruct AppleScript to replace the property only if it already exists), (3) the script context ID that's to be changed, (4) the AEDesc, and (5) the value you're setting the property to, in our case the new parent. The OSA routine OSAGetProperty performs the converse function: you can use it to inspect the values of properties and variables.

Here's a fragment from SimpliFace2 that sets the parent of a script by calling OSASetProperty:

```
OSAError    err = noErr;
AEDesc      nameDesc;
DescType    thePropCode = pASParent;

err = AECreateDesc(typeProperty, (Ptr)&thePropCode, sizeof(thePropCode),
                   &nameDesc);
if (err == noErr) {
   err = OSASetProperty(scriptingComponent, kOSAModeNull, contextID,
                        &nameDesc, newParentID);
   AEDisposeDesc(&nameDesc);
}
```

The structure of the inheritance chain is static; each parent link needs to be set up only once, as long as no scripts are replaced. The only exception to this is that the parent property of the global variables script used in SimpliFace2 needs to be set every time an incoming Apple event message is handled, as I'll explain later. Whenever a script in the chain is replaced by a new one, the script's OSAID will change and you'll need to set the parent property in the new script and in its children again.

STRIPPING COPIED PARENT SCRIPTS

By setting the parent properties of scripts and thus linking them in inheritance chains, your program limits unnecessary duplication of script objects. Still, when AppleScript sets the parent of a script, it stores a copy of the script's parent (and of the parent's parent, and so on) with the original script. This is the basis of the trick that allows SimpliFace to simulate sharing scripts between objects: every script carries with it a copy of all the scripts it shares. But this is wasteful — it means that, for instance, each button script for a window contains a copy of the window's script, when only one copy is necessary. Because your program is directly controlling script inheritance chains, you'll want to block this behavior when it loads and stores scripts. You can do it by specifying the kOSAModeDontStoreParent flag when you call OSAStore and recreating the inheritance chain when the scripts are reloaded.

Listing 1 shows the routine used to set the script property of an object in SimpliFace2. Note how it's changed from the routine used in SimpliFace: it now strips the copied parent scripts from the incoming script so that SimpliFace2 can manage the inheritance chain itself.

Listing 1. TScriptable Object::SetProperty

```
OSErr TScriptableObject::SetProperty (DescType propertyID,
                                      const AEDesc *theData)
{
   OSAError       err = errAEEventNotHandled;

   switch (propertyID) {
   case pScript:
      OSAID theValueID = kOSANullScript;
      if (theData->descriptorType == typeChar
            || theData->descriptorType == typeIntlText)
         err = OSACompile(gScriptingComponent, theData,
                     kOSAModeCompileIntoContext, &theValueID);
      else {   // If it's not text, we assume script is compiled.
         err = OSALoad(gScriptingComponent, theData, kOSAModeNull,
                     &theValueID);
         // The following new section strips any existing parent script.
         if (err == noErr) {
            AEDesc      newData;
            err = OSAStore(gScriptingComponent, theValueID,
                        typeOSAGenericStorage,
                        kOSAModeDontStoreParent,
                        kOSAModeDontStoreParent, &newData);
            if (err == noErr) {
               OSADispose(gScriptingComponent, theValueID);
               theValueID = kOSANullScript;
               err = (OSErr)OSALoad(gScriptingComponent, &newData,
                                    kOSAModeNull, &theValueID);
               AEDisposeDesc(&newData);
            }
         }
      }
      if (err == noErr) {
         if (fAttachedScript != kOSANullScript)
            OSADispose(gScriptingComponent, fAttachedScript);
         fAttachedScript = theValueID;
         err = SetCurParent(fParentObj);
         // This fixes up the references in any object that
         // has the current object as its parent.
         this->FixUpScriptReferences(this);
      }
      break;
   }
   return (OSErr)err;
}
```

ATTACH SHARED HANDLERS AND GLOBAL VARIABLES SCRIPTS

Now the plot thickens. You're going to use the inheritance chain you've set up to make it possible for your program's AppleScript scripts to share handlers and properties and to support global variables.

Our strategy, as demonstrated in SimpliFace2, is to attach a shared handlers script to the application object and a global variables script to the global script administrator object (which, as in SimpliFace, is responsible for fetching the script attached to objects and preparing it for execution). The shared handlers script, which as a convenience for the scripter I've made a property of the application object (in addition to the application object's attached script), defines common subroutines for all the object scripts known to the program. This script is added to the end of the inheritance chain so that it becomes the parent of all other scripts. Globals are created in the global variables script, which is always inserted at the start of the AppleScript inheritance chain when an Apple event is handled by a script.

Let me explain why the global variables script is necessary. The AppleScript OSA component creates global variables in the script context that received the current message, the one at the start of the inheritance chain. If the variables weren't predefined as properties when the parent script was defined, they won't be visible to all scripts. But we'd like variables that are declared in handlers with the Global keyword to be available to handlers in all scripts. We achieve this by adding the global variables script that we create to the start of the inheritance chain. Thus, the only script actually to be dispatched messages is the global variables script, and its parent becomes the currently resolved object's script. Because its parent changes to whatever is the appropriate script in the inheritance chain, messages are handled by the correct targets.

HOW IT ALL WORKS

The AppleScript-only inheritance approach demonstrated in SimpliFace2 works like this: Whenever the Apple event prehandler receives an event that might be handled in a script, it tries to resolve the object that should handle it. If it can't resolve an object it assumes that the application object's script should handle the event instead. It then attaches the global variables script (the script that receives all incoming events) to the start of the inheritance chain that ends with the shared handlers script (the parent of all scripts). In between is the script of the object to which the event was targeted; if that object is a button or a field in a window, the inheritance chain also contains the window's script. The result is the inheritance chain shown in Figure 5.

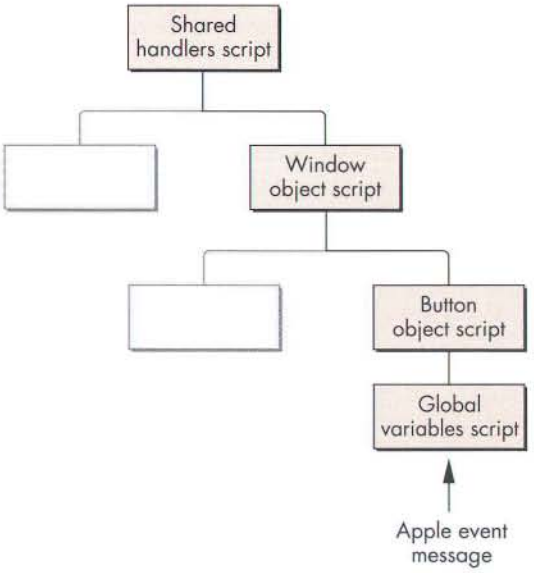


Figure 5. The script inheritance chain used in SimpliFace2

The Apple event prehandler routine in SimpliFace2 calls the global script administrator object to manage the scripts, as shown in this extract from the prehandler:

```
TScriptableObject*   theScriptableObj = NULL;
TScriptableObject*   savedParent = NULL;

if (err == noErr && theToken)
   err = gScriptAdministrator->GetAttachedScript(theToken->GetTokenObj(),
                                 theScriptableObj, savedParent);
if (err == noErr) {  // Pass to script for handling.
   if (theScriptableObj)
      err = ExecuteEventInContext(theEvent, theReply, theScriptableObj);
   else
      err = errAEEventNotHandled;
   if (theToken)
      gScriptAdministrator->ReleaseAttachedScript(theToken->GetTokenObj(),
                              savedParent);
   }
```

The script administrator function GetAttachedScript (Listing 2) is responsible for adding the global variables script to the start of the inheritance chain by setting its parent to the script of the target object. Here's how it works: First, it asks the scriptable object that's the target for the Apple event message to deliver its attached script. If there's no attached script, the parent of the global variables script is set to be the shared handlers script. If the object does have an attached script, that script becomes the parent of the global variables script.

Listing 2. TScriptAdministrator::GetAttachedScript

```
OSAError TScriptAdministrator::GetAttachedScript (
                                    TScriptableObject* theObj,
                                    TScriptableObject* &useObject,
                                    TScriptableObject* &savedParent)
{
   OSAError    err = noErr;
   OSAID       theObjScript = kOSANullScript;

   if (theObj)
      theObjScript = theObj->GetObjScript();
   if (theObjScript != kOSANullScript)
      err = StartUsing(theObj, savedParent);
   else  // If target has no script, new parent is shared handlers.
      err = StartUsing(NULL, savedParent);
   if (err != noErr)
      useObject = NULL;
   else
      useObject = this;    // If OK, return global variables script.
   return err;
}
```

GetAttachedScript returns the ID of the global variables script so that the prehandler can send the Apple event to the inheritance chain that the script now starts. The

global variables script receives the Apple event message in the function ExecuteEventInContext, called from the prehandler. The StartUsing function in Listing 3, which is inherited by the script administrator from TScriptableObject, returns the current parent of the script so that it can be saved for subsequent restoration by the script administrator function ReleaseAttachedScript.

Listing 3. TScriptableObject::StartUsing

```
OSAError TScriptableObject::SetCurParent (TScriptableObject* theParent)
{
   OSAError    err = noErr;
   OSAID       newParentScriptID = GetParentScript();

   if (fAttachedScript != kOSANullScript)
      err = gScriptAdministrator->SetScriptParent(gScriptingComponent,
                                    fAttachedScript, newParentScriptID);
   return err;
}

OSAError TScriptableObject::StartUsing (TScriptableObject* newParent,
                              TScriptableObject* &oldParent)
{
   oldParent = fParentObj;    // Returned in case it must be saved.
   return SetCurParent(newParent);
}
```

The parent of the global variables script must be saved and restored every time the Apple event prehandler handles an Apple event. This is because Apple events are dispatched recursively during the execution of scripts: you should assume that any Apple event handler can be interrupting the processing of another Apple event. For the same reason, if you need to set the resume/dispatch procedure differently in different handlers you must carefully save and restore it each time. The sample code in SimpliFace2 contains examples of how you might do this.

SimpliFace2 also shows how you can make global variables persistent, by saving the global variables script in a script file in the Preferences folder when the program exits and reloading it when the program starts up again. The code to handle this is in the script administrator routines SaveGlobalVariables and LoadGlobalVariables.

A RUN HANDLER WRINKLE

Be aware of a wrinkle: If you ever intend to dispatch the Run ('oapp') Apple event to an AppleScript inheritance chain, each script in the chain must contain a continue run handler.

```
on run
   continue run
end run
```

The reason for this is that when AppleScript compiles a script into a script context it collects all the top-level statements in the script (those not contained in any handler) into a default run handler, so that if the script is simply executed the top-level statements will run. If there are no top-level statements, an empty run handler is created. The trouble is, this default handler doesn't realize you want it to continue

the Run message, so the message will be caught and lost in the script, never to be seen farther along in the inheritance chain.

INHERITANCE IN OTHER OSA LANGUAGES

The inheritance scheme just described is specific to AppleScript scripts because it relies on OSASetProperty to set up the inheritance chain. This call and others in the AppleScript 1.1 API aren't part of the required OSA API, so not all scripting components support them. If your program is to support inheritance in scripts written in other OSA languages, it must take control of the message passing between scripts in the chain at script execution time.

Your application can do this by simulating, entirely under program control, the mechanism that AppleScript uses. Messages that aren't handled in a particular script in the chain are passed along to the next script in the chain. If they're continued out of, or not handled in, the chain, the program routes them to its standard Apple event handlers. The drawback to this approach is that scripting becomes a little more rigid, because handlers are resolved only toward the parent rather than wherever they are in the chain. The advantage is that it will work with any OSA language that supports the event handling API and the Subroutine Apple event, which is the message protocol AppleScript uses to call subroutines.

When your program's Apple event prehandler deals with an incoming Apple event message by passing it to a script, it's responsible for manually redispatching messages that aren't handled or that are continued along the inheritance chain. The SimpliFace2 routine ExecuteEventInContext (Listing 4) shows how to do this. The first and second parameters to this routine are the Apple event and reply; the third parameter is the scriptable object that is to handle the message. This routine is first called from the prehandler, which passes it the scriptable object that starts the inheritance chain (the object to which the message was originally sent). The scriptable object has a field that's a reference to its parent object, which can be read using the accessor function GetParentObj. If the current object has a parent, the routine sets the OSA resume/dispatch procedure to be a recursive call to ExecuteEventInContext, passing the address of the parent object as the reference constant. If the current object doesn't have a parent (if the end of the chain has been reached), the routine sets the resume/dispatch procedure to be the program's standard Apple event handler, ignoring the prehandler.

If the Apple event message isn't handled in the script of the current object, the routine OSADoEvent returns the error errAEEventNotHandled. At this point you must manually redispatch the message, mirroring the OSA's resume/dispatch mechanism: if the current object has a parent, you recursively call ExecuteEventInContext, passing it the address of the parent object. If you've reached the end of the inheritance chain, you simply call the program's standard Apple event handler.

WHERE YOU'VE BEEN, WHERE YOU'RE GOING

You've learned that implementing inheritance starts with choosing an appropriate script inheritance hierarchy. With this hierarchy in mind, you can link scripts in an inheritance chain, either by setting their parent properties (if you're working only with embedded AppleScript scripts) or by directly controlling inheritance at script execution time (if your program needs to support scripts in other OSA languages). Then you're ready to create a shared handlers script and put it at the end of the inheritance chain, making it the parent of all scripts, and (if you're working only with AppleScript scripts) to add a global variables script to the start of the inheritance chain where it can receive incoming messages and route them to the correct targets.

Listing 4. ExecuteEventInContext

```
static pascal OSErr ExecuteEventInContext (AppleEvent *theEvent,
                            AppleEvent *theReply,
                            TScriptableObject* theScriptableObj)
{
   OSAError    err = errAEEventNotHandled;
   OSAID       theScriptID = kOSANullScript;

   if (theScriptableObj)
      theScriptID = theScriptableObj->GetObjScript();
   else if (gScriptAdministrator)
      theScriptID = gScriptAdministrator->GetSharedScript();
   if (theScriptID != kOSANullScript) {  // Pass to script for handling.
      AEHandlerProcPtr        oldResumeProc;
      long                    oldRefcon;
      TScriptableObject*      parentObj = NULL;
      OSAGetResumeDispatchProc(gScriptingComponent, &oldResumeProc,
                               &oldRefcon);
      if (theScriptableObj) {
         parentObj = theScriptableObj->GetParentObj();
         err = OSASetResumeDispatchProc(gScriptingComponent,
                        (AEHandlerProcPtr)&ExecuteEventInContext,
                        (long)parentObj);
      }
      else
         err = OSASetResumeDispatchProc(gScriptingComponent,
                        kOSAUseStandardDispatch, kOSADontUsePhac);
      if (err == noErr) {
         err = OSADoEvent(gScriptingComponent, theEvent, theScriptID,
                        kOSAModeAlwaysInteract, theReply);
         if (err == errAEEventNotHandled) { // Not handled in script.
            if (theScriptableObj)   // Make recursive call.
               err = ExecuteEventInContext(theEvent, theReply, parentObj);
            else  // Otherwise, dispatch directly to standard handler.
               err = StdAEvtHandler(theEvent, theReply, 0);
         }
      }
      OSASetResumeDispatchProc(gScriptingComponent, oldResumeProc,
                               oldRefcon);
   }
   return (OSErr)err;
}
```

You've seen these techniques illustrated by the sample program SimpliFace2. You can simply drop the classes from SimpliFace2 into your own programs, use them as the basis of new programs, or use them as a guide to restructuring existing programs. Armed with the information in this article and its predecessor, you should now be able to implement an inheritance scheme for scripts in your own software.

Thanks to our technical reviewers Kevin Calhoun, Ron Karr, and Jeroen Schalk, and to Lee Buck of Software Designs Unlimited.•

Macintosh Q & A

Q *What operations cause QuickDraw GX to invalidate its shape caches? We want to maximize drawing performance in our application, and it would be nice to know ahead of time when caches will need to be rebuilt.*

A As you might imagine, this topic is complex, but we'll try to explain the basics and give you some ideas about improving performance.

There are several caches associated with a shape, since each object associated with the shape has a separate cache. (By the way, the caches are in the QuickDraw GX heap and can be seen using GraphicsBug.) Every time you call GXDrawShape, QuickDraw GX determines which caches need to be updated, updates them, and then draws the shape. So the first thing to know is that if you want GXDrawShape to execute as fast as possible (for instance, if you're drawing several shapes and want the time between successive GXDrawShape calls to be minimal), you should call GXCacheShape ahead of time to update the shape's caches, minimizing the work GXDrawShape needs to do.

In general, any time you change information within a shape, you cause QuickDraw GX to invalidate at least one of the shape's caches. Layout shapes are exceptions to this rule, however. The cache associated with a layout shape will not be invalidated if all of the following conditions are met:

- You're adding characters to the end of the layout shape.
- There's a single run of text within the layout shape.
- The shape remains on the same device.

If you're drawing non-hairline geometric shapes and want to get them on the screen as fast as possible, you can set the gxCacheShape attribute of the shape. With this attribute set, QuickDraw GX will preprocess all the required parts of a shape into compressed bitmaps. This means the shape is completely ready to be drawn when you call GXDrawShape: the bits are just blasted to the screen.

Any time you change a view port's clip or mapping, QuickDraw GX will update all of the caches for the shapes and child view ports associated with that view port. There isn't any speed advantage to setting the clip or mapping of a shape rather than the clip or mapping of the shape's view port; the same work has to happen in either case. Note, though, that if the view port contains other shapes or has child view ports, their caches will be invalidated, too. You should change the view port's mapping to move a shape only when you want all shapes within the view port to move the same amount.

Q *I'm looking into the possibility of writing a QuickDraw GX printer driver that will only print to a file, and I've run into a couple of stumbling blocks I hope you can help with. First, is there a way to create a desktop printer from the Chooser without having an actual printer attached or selecting a port? Can I get to the Chooser list so that I can display some kind of dummy or ghost printer? The second issue has to do with the user interface of the Print dialogs/panels. I would like to set the "Destination: File" radio button and disable the "Destination: Printer" one for QuickDraw GX application interfaces.*

A There should be no problem creating a printer driver that only prints to a file. And yes, users will be able to create a desktop printer from such a driver. You can access the Chooser list: for an example of this, look at the source code file Chooser.c from any of the QuickDraw GX printer driver samples. You can do

whatever you want within the Chooser to handle and display lists of available printers for the currently selected printer driver. You'll also need to modify the 'comm' resource to make sure that the Chooser does the right thing. You should be able to put a "nops" 'comm' resource reference into the 'look' resource.

You can disable any item in the Print dialog by overriding the GXJobPrintDialog message and, in your override, getting the destination tag for the appropriate item and locking it. This will make the item appear disabled in the dialog. PrintingMgr.h contains all the tags you can lock. You should be sure to set up the item to be gxVolatileOutputDriverCategory so that your settings go away if the user switches drivers in the pop-up menu; you can simply OR this with collectionLockMask when you set the item's attributes.

Q *The documentation seems a bit thin on what resource type/ID is needed for ColorSync profiles in QuickDraw GX printer drivers. Is the appropriate type 'prof' or 'cmat'? Will QuickDraw GX automatically use one if I stick it in my driver, or do I also have to override the various profile messages?*

A All you need to do is include a 'cmat' resource with ID gxColorMatchingDataID and specify it in your 'rdip' resource. However, you should also override GXFindFormatProfile so that inquiring applications can ask you what the format's profile will be. Additionally, you should override the GXImagePage message if you want to provide different profiles on a page-by-page basis.

Q *I'm working on QuickDraw GX printer drivers. Can you give me some information on what must be done to support PrGeneral?*

A PrGeneral support within your QuickDraw GX printer driver is automatic. The printing system will automatically maintain the appropriate information within the QuickDraw GX print record. The only exception to this automatic support is the SetRsl opcode: if you don't want QuickDraw GX to use the default gxReslType resource when the SetRsl opcode is used, you need to define a gxReslType resource of your own that reflects the capabilities of your printer.

Q *Our application generates its own custom PostScript™ code when printing to PostScript printers. We'd like to support QuickDraw GX–style printing and still be able to continue generating our own custom PostScript code to send directly to the printer. What's the best way to handle this? I've tried generating empty shapes with custom PostScript code attached as a tag (using synonyms), but the LaserWriter QuickDraw GX driver emits its own "wrapper" code around our custom PostScript code, which could alter the printer's graphics state. Are there any other methods to achieve this without the possible side effects?*

A You're actually very close to sending PostScript code correctly through the QuickDraw GX printing system without the side effects. As you already know, after a shape with 'post' tags has been sent to a PostScript printer, QuickDraw GX performs a PostScript restore. There's no way to prevent this from happening. However, only shape objects cause this behavior; QuickDraw GX will not perform a PostScript restore for any other object besides a shape.

The fastest and best method for sending PostScript code is to attach the code with tags (using synonyms) to only one empty shape. You can attach as many

tags as are required. We recommend that the tags contain 12K of PostScript data each, for optimal performance. If you're sending PostScript code down to replace the clip or ink or some other object besides a shape, just attach a 'post' tag to the object your PostScript code describes; in this case a restore will not occur.

Q *How should I download fonts to a PostScript printer under QuickDraw GX? I'm sending a direct stream of custom PostScript code, but I don't expect the driver to be able to deduce which fonts need to be included. I've read about the PostScript control tag that can be attached to a shape, and I know the structure for the tag contains font information, but the documentation about this tag is sketchy. Can you provide more details?*

Related to this question, what's the best way to cause fonts to be downloaded under the current Printing Manager? We're using the "draw a single character off the page in the proper font" trick. I understand this practice is frowned on. Is there an approved way to do this short of intermixing QuickDraw and PostScript code in PicComments?

A Our thinking has changed regarding the use of the PostScript control tag for downloading a font. If you want to download a font within your PostScript stream, you should call GXFlattenFont and pass the font within your 'post' tag. The GX PostScript engine will unflatten the font and download it when it finds it attached to your 'post' tag.

The method you're using (drawing a character off the page) is completely supported under QuickDraw GX. It was the recommended method for a long time. However, the current recommendation is to use DrawText and pass in text that contains all the glyphs you want to use. The reason this approach is better for QuickDraw GX is that only the required information is downloaded, thereby saving memory on the printer.

Q *Do I need to override the GXFreeBuffer message in my QuickDraw GX printer driver if I perform a total override of the GXDumpBuffer message? If I do need to override GXFreeBuffer, what do I do with the buffer? Should I dispose of it with DisposePtr?*

A The documentation is a little confusing about the purpose of the GXFreeBuffer message. GXFreeBuffer is sent to ensure that the indicated buffer has been processed and is now available for more data; it doesn't actually dispose of a buffer. The only time you need to override GXFreeBuffer is if you're doing custom I/O (the customIO flag is set in your 'iobm' resource). The default implementation of GXFreeBuffer will work correctly for QuickDraw GX's default buffering mechanism.

GXFreeBuffer allows you to start asynchronous I/O in GXDumpBuffer; then other buffering routines can be sure that operations on a buffer are completed before they reuse the buffer (or dispose of it). If you're overriding GXFreeBuffer, you should just hang out in your override until I/O has completed for the buffer passed, and then return. If you're doing synchronous I/O, just return immediately, since all I/O must have completed.

Q *When I control the start of a QuickTime movie from within my application, the movie controller doesn't get updated properly. I'm calling StartMovie to begin the movie as soon as it becomes visible, and I'm updating the movie controller like this:*

```
theResult := MCDoAction(tmpMovie.fController, mcActionPlay, @curRate);
```

However, this doesn't seem to work. What am I doing wrong?

A The MCDoAction call with mcActionPlay doesn't take a pointer to the data in the last parameter; it takes the data itself. But since the prototype specifies type (void *), to make the compiler happy it must be cast to a pointer. Many people have fallen into this trap.

The recommended method to start a movie when you're using the standard movie controller component is as follows:

```
// Play normal speed forward, taking into account the possibility
// of a movie with a nonstandard PreferredRate.
Fixed rate = GetMoviePreferredRate(MyMovie);
MCDoAction(MyMovieController, mcActionPlay, (void *)rate);
```

If you do need to use StartMovie, the correct way to cause the movie controller to update is to call MCMovieChanged.

Q *We're using the Communications Toolbox in our application and have noticed some strange behavior. Specifically, when a tool is being used, the tool's resource fork is placed not at the top of the resource chain, but behind the System and application resource forks. As a result, we're having trouble with tools that use STR# resources that conflict with those in our application. This results in bad configuration strings or configuration dialogs with the wrong text. I can easily work around this by changing the resource IDs in my application, but this doesn't solve the problem in the long run. Any advice?*

A What you're seeing is a part of the "pathology" of the Communications Toolbox and its tools. Both do a less than perfect job of looking in the correct resource map for string resources. The result is what you're currently seeing: application strings end up being placed where tool strings are expected.

There are a couple of workarounds. The first you've cited, which is to make sure that none of your application's resource ID numbers conflict with the CTB tool's ID numbers. However, as you also noted, this can be a problem when you're using several CTB tools, and may not work if the user happens to select a tool that has a conflicting ID.

The better way to work around the problem is to save a reference to the current resource file and then set the resource file to the System file. After completing a call to CMChoose or CMGetConfig, you can reset the resource file to the one you started with. Here's how to do this:

```
short    oldResFile;
Point    dlogBoxPt;
Ptr      myConfigString;

/* First save the current resource file. */
oldResFile = CurResFile();
/* Now set the resource file to the System file. */
UseResFile(0);
/* Next call CMChoose to configure your connection tool. */
myErr = CMChoose(&myConnectionHdl, dlogBoxPt, nil);
```

```
/* Now call CMGetConfig to get the configuration string from the
   connection tool. */
myConfigString = CMGetConfig(myConnectionHdl);
/* And finally, restore the old resource file. */
UseResFile(oldResFile);
```

Q *What is the Human Interface suggestion for removing a digital signature from a signed document? I see how to add and how to verify, but I can't find any suggestions for removing signatures.*

A Here's the relevant paragraph from the *AOCE Human Interface Guidelines* document, which can be found on AppleLink (search the AOCE Talk folder):

> Signatures may also be deleted by users. To accomplish this, the user should select the signature by clicking on its icon and choosing Clear from the application's Edit menu. Selecting the signature icon and pressing the delete key is a desirable alternative. Note that signatures must not be cut, copied, or pasted.

To actually remove a signature, just remove the 'dsig' resource from the file. Note that signed files may have the Finder "locked" bit set. If you remove the signature, you should also clear this bit.

Q *I launch my application by dragging files onto its icon. It then opens the files, performs some quick operation, and quits. I can put '****' and 'fold' resources in FREFs to let users drag any file or folder onto the application icon. But when a user drags an AOCE catalog (or anything inside the catalog) onto the icon, the Finder won't let the user drop it onto my application. What do I need to do to my application to let users drop-launch it with AOCE catalogs (or their contents)? I know it's possible: the "Find in Catalog" application will drop-launch if a user dragged to it from a catalog.*

A If you look, you'll see that "Find in Catalog" is a Catalogs Extension template. It's not an application program; it actually executes as part of the Finder. Unfortunately, you can't do what you want to with an application. You might want to look at the Catalog Service Access Modules chapter in *Inside Macintosh: AOCE Service Access Modules* for more information on the Catalogs Extension.

Q *When users launch my utility application by double-clicking, I present a Standard File dialog that lets them choose a file to operate on. I'd like them to be able to browse AOCE catalogs as well as HFS files, but catalogs don't show up in the Standard File dialog. I could use another dialog for browsing AOCE catalogs, but why use two different interfaces for the same action (from the user's point of view, that is; the user just wants to specify an object, wherever it may be)? Is there a way to get catalogs to show up in the Standard File dialog? Is there any way to browse the file system and the AOCE catalog system in the same dialog? If the answer is no, is there an analog to Standard File for AOCE catalogs?*

A You can't browse HFS files and AOCE catalogs in the same dialog, since they're two different file systems. To let the user browse the AOCE catalog system, you need to use the AOCE Standard Catalog Package Reference routines, which are documented in *Inside Macintosh: AOCE Application Interfaces*. There is a routine that's analogous to StandardGetFile. The AOCE Software Developer's Kit

(available through APDA) includes sample code that shows how to browse AOCE catalogs.

Q *I'm writing an application that will watch a user-specified folder and operate on files or folders that are dropped into it. I need to perform operations on every item in the folder and its subfolders. What will happen if I begin to walk through a new folder with PBGetCatInfo while the Finder is still copying files into the subfolder structure? Can I guarantee that I'll detect all the files?*

A There's no way to find out directly when the Finder is done copying items into the folder, but there is a strategy we can recommend. First, though, you should know that the recommended way to determine whether items have been added to a folder is to watch its modification date. So how can you know when all the files have arrived? The recommended strategy is to poll the parent folder's modification date every few seconds after you first detect a change, and keep polling until you have a reasonably long interval during which there's no change in the date (30 seconds is probably about right). This means, of course, that there's a delay before you act on files dropped into the folder, but as the Print Monitor shows us, this delay is probably reasonable to the user.

One more possible gotcha you should know about: to tell whether it's safe to work with a particular file, it's not enough just to make sure the file isn't open; you need to check its length. If a file is closed (that is, not busy) but has no length in either its resource or data fork, you caught the Finder at an uncomfortable time, after it created the file but before it opened it. So to know if you can operate on a file the Finder might be copying, you must determine two things: it's not already open, and it has length in its resource or data fork.

Q *Our application uses the Icon Utilities interface, and we want to verify that these features are present before we use them. We've been unable to do this successfully. The Macintosh Technical Note "Drawing Icons the System 7 Way" (QuickDraw 18) doesn't say how to do this, but Inside Macintosh: More Macintosh Toolbox recommends using Gestalt with the gestaltIconUtilities selector. When we try this, Gestalt returns an error of -5551 (undefined selector). What are we doing wrong?*

A There isn't a Gestalt selector for the Icon Utilities; *Inside Macintosh* and the header files are wrong. Even if we were to correct that situation tomorrow (or in the next system software release), it wouldn't help, since the Icon Utilities are available on systems where the Gestalt selector isn't. The solution is to use the TrapAvailable function to see if the _IconDispatch A-trap is available. You can find the source code for TrapAvailable in *Inside Macintosh* Volume VI on page 3-8, or in *Inside Macintosh: Overview* on page 180.

Q *I have a System 7 application that the user can drag files, disks, or folders to. How can I determine from the Apple event information which type of item (file, disk, or folder) has been dragged to the application icon?*

A When the user drags a file, disk, or folder to an application icon, the Finder uses the Process Manager to open the application and then sends it an Open Document ('odoc') event containing a list of alias records for each object dropped. When your application receives the event, it needs to open each of the objects specified in the event by getting each alias record from the list and coercing the data for that record to an FSSpec. Once you have the FSSpec, you

can check its parID to determine whether the directory ID is fsRtParID, indicating that you're looking at a volume. If the parID is anything other than fsRtParID, use PBGetCatInfo to determine if you have a file or a folder. Here's the code:

```
enum {kItsAVolume = 1, kItsAFolder, kItsAFile};
pascal OSErr GetSpecType(FSSpec *myFSS)
{
   CInfoPBRec  pb;
   OSErr       myErr;
   short       objType = 0;

   if ((myFSS->parID) == fsRtParID)
      objType = kItsAVolume;
   else {
      pb.hFileInfo.ioNamePtr = (StringPtr) *(myFSS).name;
      pb.hFileInfo.ioVRefNum = *(myFSS).vRefNum;
      pb.hFileInfo.ioDirID = *(myFSS).parID;
      pb.hFileInfo.ioFDirIndex = 0;
      myErr = PBGetCatInfoSync(&pb);
      if (myErr == noErr) {
         /* Check to see if bit 0x10 of ioFlAttrib is set; if it is,
            we've got a directory */
         if ((pb.hFileInfo.ioFlAttrib & 0x10) != 0)
            objType = kItsAFolder;
         else
            objType = kItsAFile;
      }
   }
   return (objType);
}
```

Q *I'm trying to implement "the perfect component." The goals for this component are fast dispatching, delegatable, able to rely on delegates, ready for everything, and surprised by nothing. I've been using* develop *Issues 12 and 14 and Inside Macintosh: More Macintosh Toolbox to guide me, but I still have a question that wasn't addressed in those references.*

I'm using the Fast Dispatch method to dispatch my component's calls. I've figured out how to repair the stack after getting an unsupported routine selector code, but I can't figure out how to delegate a call. I think I'm recovering the stack correctly, but all the documentation I've read doesn't even hint at this sort of functionality. I was thinking that I could use DelegateComponentCall after creating a ComponentParameters record, or I could calculate the size of the parameters and attempt to set up the stack for a ComponentCallNow call. However, neither of these is a good solution — they both take too many instructions to implement and obviate the advantages of fast dispatching. Is there such a thing as a DelegateFastComponentCall that I haven't heard of?

A Try the following to delegate a component call:

```
move.l   d0, -(sp)      ; push d0 onto stack
PUSHDELEGATEGUY         ; macro to push component instance
move.q   #-2,d0
dc.w     $a82a
```

What apparently happened is that the engineers realized how difficult it was to call DelegateComponentCall when the stack was screwed up. So they created a "special" delegate call. As you can see, the selector is -2. This is supposed to be documented somewhere in the Component Manager documentation, but was inadvertently left out.

Q *Why aren't my components getting register calls? I've set the appropriate flag. I'm registering the component from my application.*

A Your component won't get called to register at all since it's being registered by an application. If you removed your component and placed it in the Extensions folder so that it was registered at system startup, it would receive a register call. The reason for this is that registration of components happens only when the Component Manager is first loaded at system startup. After system startup, components can be registered by applications, but the register routine will not be called.

Q *I heard that the new Apple digital camera will introduce a graphics format called QuickTake. Is this truly a new format or are these just QuickTime compressed PICTs? If it's a new format, where can I find documentation on it?*

A QuickTake stores its pictures as compressed PICT files. There's a new CODEC that's necessary to decompress the images, but no new file type. The QuickTake 100 Digital Camera Developer Note contains extensive information about talking to the camera from both Macintosh and Windows machines. It also documents the picture formats. The QuickTake Camera Software Development Kit is available from APDA.

The QuickTake application that comes with the camera can save the pictures in a variety of formats, including PICT (with or without various compressions) and TIFF. There's also a control panel that allows the user to mount the camera as a read-only serial RAM disk (similar to MountImage), so your application can directly download the information from the camera's memory.

Q *I'm writing my first action atom for the Installer. I tried writing a skeletal example, as follows:*

```
resource 'inaa' (30000) {
   format2 {
      continueBusyCursors, actAfter, dontActOnRemove,
         actOnInstall,
      'infn', 149,
      0,
      1000,
      "Delete Folder"
}};

#include "ActionAtomHeader.h"

ActionAtomResult ActionAtomFormat2(ActionAtom2PBPtr aa)
{
   Debugger();
   return (kActionAtomResultContinue);
}
```

But the action never gets called! I don't have to explicitly refer to the 'inaa' anywhere, do I? I checked that the resources were copied into the file correctly and had the right IDs. What am I doing wrong?

A Your action atom is fine. The only mistake you've made is not tying your 'inaa' to a package ('inpk'). You have to add to your 'inaa' a reference to an active 'inpk'. Once you do that, it works great. (We had trouble with this one until we found a cool diagram on page 8 of the Installer documentation that shows how the script resources interrelate; that diagram is your friend, and seems to encapsulate quite a bit of critical information.)

Q *We're planning a zone name change for the zone that contains some of our PowerShare servers. Will this require any action on the part of the administrators of the various servers beyond informing users of the change?*

A It depends on whether you're changing the zone names of some or all of the servers. If you're changing the names of only some servers, it's entirely possible that you don't have to notify anybody, depending on how you've replicated your folders. For a while the new servers won't be contacted by any clients since all clients will try to address them at their old location, but eventually all servers will again start receiving requests from clients.

If you're changing the zone names of all the servers, the client software won't recognize that the zone names have changed, so you'll have to throw away your key chain and add it again after the zone names have changed. The PowerShare servers will eventually recover and rediscover all the other servers. We recommend that you bring up the Master Pathfinder first after changing its zone name and then bring up all the other servers. Once all the servers are up, it should take a few hours at most for everything to settle down again and for the system to be purring.

Q *The function SMPEnumerateBlocks takes a buffer that returns information about the blocks in a letter. The documentation (Inside Macintosh: AOCE Application Interfaces, page 3-87) says that the buffer contains "a count byte indicating the number of blocks in the letter, followed by a block information structure for each block." I can't find anything that tells me what a "block information structure" is.*

It looks to me as if it actually returns the count of blocks in a short, not a byte. Is this the total number of blocks, or the number of blocks returned by the call? If I have to call the function again to get information that didn't fit in the buffer, do I get another count followed by more block information structures, or do I just get an array with more block information structures without the count (short)?

The block information structure seems to be 16 bytes long, starting with an OCECreatorType structure. Is this always correct, and what is the other information?

A This is somewhat confusing. The "block information structure" referred to here is the MailBlockInfo structure defined in the OCEMail.h header file:

```
struct MailBlockInfo {
   OCECreatorType    blockType;
   unsigned long     offset;
   unsigned long     blockLength;
};
```

And yes, the count byte returned in the buffer you provide is actually a short (the documentation is wrong). This value is placed right in front of the first MailBlockInfo structure each time you make the call. The count indicates the number of blocks that were put in your buffer for this particular call (not the total), which of course depends on the size of the buffer you pass; it fits in as many as it can. Check the "more" parameter to see if your buffer was too small to hold all the blocks, and check the nextIndex parameter for the sequence number of the next item to be returned.

Q *I found what I think is a problem with the TEGetOffset routine: when it returns, a 28-byte handle has been locked for single-styled text. I believe it's a style handle that gets locked. This seems to be an intermittent bug, occurring only about half the time. Is there a workaround? Is it safe to just unlock the style handle after calling TEGetOffset?*

A It's a bug all right, and it is the style handle that's getting locked. Here's a workaround:

```
static short MyTEGetOffset(Point pt, TEHandle th)
{
   TEStyleHandle  sh;
   short          theResult;
   char           saveState;

   if (th == 0L)
      return;
   sh = GetStyleHandle(th);
   saveState = HGetState(sh);
   theResult = TEGetOffset(pt,th);
   HSetState(sh, saveState);
   return theResult;
}
```

Q *I have a question about the Japanese art of bonsai (colloquially known as stunting trees): What happens if you bonsai a fruit tree? Does the fruit come out dwarfed as well? Or does the tiny tree produce full-size fruit?*

A This was a difficult one to find an answer to. It seems to depend on many factors, not the least of which is the kind of fruit tree you're talking about. One person we talked to said he once saw a bonsai lemon tree nine inches tall, with a single, full-size lemon hanging from a branch. (Actually, the lemon didn't hang; it would have broken the poor tree. Instead it rested on a small platform built especially for that purpose.) But there were also reports of a stunted crabapple tree that produced apples the size of peas. Go figure.

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Pete ("Luke") Alexander, Joel Cannon, Mark ("The Red") Harlan, Dave Hersey, Dave Johnson, Don Johnson, Scott Kuechle, Jim Luther, Kevin Mellander, Jim Mensch, Martin Minow, and John Wang for the material in this Q & A column. If you need more answers, take a look at the Macintosh Q & A Technical Notes on this issue's CD.•

THE VETERAN NEOPHYTE

Rubber Meets Road

DAVE JOHNSON

I've been thinking about edges lately — about the places where dissimilar domains meet and interact. You know how every now and then you come up with a new view on things? A new model to try to fit the facts into, a new lens to use to examine the world, a new pattern that you haven't noticed before but that suddenly seems pervasive? Edges are like that right now for me. It seems that everywhere I look I see edges, and the edge always seems to be where the action is.

I think it started in January, when I was called for jury duty. I was promptly selected to serve on a long, complex, and sordid criminal trial. I've been called for jury duty only once before, and that time the experience was short and dull. I did serve on two juries, but neither trial lasted more than a couple of days, and they were both very mundane. This time was decidedly different. There were 4 defendants, 53 separate counts to decide, 3 different crime scenes, dozens of spent bullet casings and slugs and shotgun waddings to keep track of, something like 14 police witnesses and 6 or 8 civilian witnesses, a two-inch thick stack of 8 by 10 color glossies, and lots more. The whole adventure took nine weeks to play out. Yow.

The atmosphere in the courtroom spanned the full range of intensities. There was plenty of plodding boredom: day after somnolent day of slow, thorough, painstaking ballistics testimony, matching bullets to guns and mapping where they were found. There was high drama: the tapes of the police transmissions during the chase and as the final shootout began were filled with panic, screaming. There was humor: Helen in chair 5 often started to fall asleep in the afternoons. The court reporter would see her dropping off, make a little hissing noise, and Wes in chair 4 would surreptitiously nudge Helen back to consciousness. We'd all grin.

But no matter what was happening at the moment, I found the *process* absolutely riveting, from beginning to end. Here were the great and mighty wheels of justice in America, slowly and ponderously turning, grinding away at the facts like so much dry corn under a millstone. The courtroom is a place where politics actually collides directly with people's lives, through the strange intervening filter called law. It's an edge, an active boundary separating two domains, where work actually gets done.

I'm constantly drawn to active boundaries like that, places where two dynamic systems collide and affect each other. Interfaces. Precipices. Limits. Edges. And they are everywhere. In a previous column I pointed out an edge in the realm of language: semantics, where a language's structure collides with meaning and where the real work of the language — creating meaning from abstract symbols — gets done. In biology, there are edges all over the place. An obvious and important one is the semi-permeable membrane. It's the structure that allows life to create and control its own environment, and it's arguably the single most important structure enabling complex multicellular life to exist. Biologically active molecules are active because of their *shape*, their boundaries; proteins and enzymes work because they fit together with complementary molecules. In philosophy there is the edge between self and not-self, and teetering along this edge, hopping back and forth across it and trying to look at it from all angles, is how the work of philosophy gets done. In physics, often the edges are where the truly interesting — and, not coincidentally, mathematically intractable — stuff happens. (In engineering school, an all-too-common phrase was "ignore edge effects.")

All the exciting stuff seems to happen at edges. Large systems that incorporate feedback often exhibit a behavior known as "self-organized criticality" in which they evolve toward a critical state, an edge, and forevermore exist there, teetering on the crumbling lip of stability. A great example is a conical pile of sand on a circular plate, with grains being added to the top one at a time. Over time the overall shape of the pile will change very little, but if you turn up the magnification and look closely at the side of the pile, there are constant avalanches of all sizes, all extremely unpredictable and chaotic. This is an interesting dual behavior: at one scale there is incredible robustness; the overall shape of the pile is very stable and will always recover itself, even if disturbed. But on a smaller scale, the scale of an individual grain on the side of the pile, the dynamics are wildly unpredictable and incredibly

DAVE JOHNSON likes to try to slip new words he's learned into casual conversations without anyone really noticing. Two years ago he learned the word *enantiomorph*. As you might imagine, he's still waiting for the right opening.•

*un*stable. The pile is poised at a limit, a dynamic balance between growth and decay.

An interesting thing is how many different varieties of dynamic systems seem to exhibit this kind of behavior. The locations and magnitudes of earthquakes, fluctuations in traffic flow, the rise and fall of economic markets, the rhythmic variations in a heartbeat, the varying current through a resistor, and the population changes in an ecosystem all exhibit dynamic characteristics similar to the sand pile, and this is not an exhaustive list by any means. That state, pushed up against the edge of stability, seems to be a natural one. Life itself appears to be delicately poised on the boundary between order and chaos.

In computers (you knew I was going to get around to this eventually, didn't you?), as in any complex system, there are lots of interesting edges and boundaries if you look for them. Internally, there's the place where the software collides with the hardware; sparks really fly down there, all right. Object-oriented programming is all about repackaging the boundaries between and among data and functions. (A large part of good object design is minimizing the "surface area" of your objects.) And then there's the edge of the computer itself. And I don't mean the plastic or metal surface of the box, but the *experiential* boundary, the true edge between the machine and the user, the interface. Here the animal collides with the machine, and the boundary between them is infinitely convoluted, elastic, dynamic, and interesting.

For software designers, perhaps the most important lesson to be learned from the edge-centric view is this: the shape of a boundary defines the shape of things on *both* sides of the boundary simultaneously. The boundary of my dog Natty defines not only her own shape, but that of a Natty-shaped hole in the air as well. The edge between two interlocking tiles in an Escher drawing defines the shape of both tiles at once. If the edge in question is one we have control over, this can be very important.

By programming a computer we're not only shaping the machine; we're also shaping the humans who use it. This is often overlooked, but is crucial to designing good software; it needs to *fit*. Humans are incredibly adaptable, and will contort themselves grotesquely to use awkward tools, if necessary. Like kids with their faces squashed against the toy store window, computer users smash themselves up against the interface — even though it might hurt — to get at what's inside.

But because of the chameleon-like nature of the computer, we have more or less total control over the interface. So in principle we have the power to shape the computer to the user, rather than the other way around. We should be able to make a truly human-shaped dent in the computer, a dent people can slip into effortlessly and comfortably, like slipping into a fuzzy slipper. It's incredibly hard work, shaping the computer to the human, all that snipping and tucking and smoothing. It requires constant readjustment, painstaking attention to detail, and massive amounts of brute-force trial and error. But it's *good* work, some would say the work that humans are best at: the shaping of tools.

So now here I am, seeing edges everywhere. Sigh. Last year it was basins of attraction, this year it's edges, next year maybe it'll be networks of interconnections. But there's one thing I can count on: every time I get tired of looking through one particular glass, there will be another within reach. Humans have this uncanny ability to apply order to everything they see, to perceive structure in everything around them. Our minds seem to operate by forming and then reforming meaning, establishing and then reestablishing context, constantly slipping and adjusting to accommodate the relentless stream of input. Hmm. Just like that pile of sand.

RECOMMENDED READING

- *Complexity: The Emerging Science at the Edge of Order and Chaos* by M. Mitchell Waldrop (Simon & Schuster, 1992).
- *How Dogs Really Work!* by Alan Snow (Little, Brown and Company, 1993).

Thanks to Jeff Barbose, Michael Clark, Michael Greenspon, Brian Hamlin, Mark ("The Red") Harlan, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their always enlightening review comments.•

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe.•

Newton Q & A: Ask the Llama

Q *I'm having trouble with the protoRoll. I have a protoApp with a protoRoll at the bottom with a couple of items in it. (Note that I'm not using the protoRollBrowser proto.) It compiles OK, but when I download to the Newton, nothing shows up. In fact, when I use the inspector to look at the view hierarchy, the protoRoll doesn't show up at all. The other views are fine. What am I doing wrong?*

A The protoRoll doesn't show up because of the setting of the viewFlags of the ROM prototype: the vApplication and vClipping flags are set, but not the vVisible flag. If the protoRoll were the base template of your application, the vApplication flag would be sufficient to make it visible.

In your case, the protoRoll is a child of your base application template. Since it isn't visible (vVisible isn't set), the system doesn't create a runtime view frame for the child. You could get the system to create the runtime view by declaring the protoRoll to be the base template, but this still wouldn't show the protoRoll.

To make the protoRoll visible, add a viewFlags slot to the protoRoll and check the vVisible flag. You may or may not want to uncheck the vApplication flag. If you uncheck it, the system will no longer send scroll and overview messages (viewScrollUpScript, viewScrollDownScript, viewOverviewScript) to the protoRoll, so it will appear to be broken. But you can support these messages in your base application view and just pass them on to the protoRoll as needed. If you leave the vApplication flag checked, protoRoll will get the scroll events.

Q *My print format never seems to get called, ever. I don't get a printNextPageScript or even a viewSetupFormScript. I'm not using ROM_coverPageFormat because I don't ever want to print a cover page. How can I get this to work?*

A The answer to your problem is in your question. A print (or fax) format must proto to ROM_coverPageFormat; it's not optional (as the manual implies). It may help to know that ROM_coverPageFormat is really misnamed. The generation of a cover page is controlled by a slot in your format. The proto should be called something like ROM_allThePrintingAndFaxingBehaviorProto, but that would be verbose :-)

Q *I would like to add a [button | view | Llama] to the [Notepad | Calendar | Cardfile | etc.]. How can I do that safely?*

A This is a simple one: you can't. If you add any element to a built-in application, you take the chance that your application will break in future releases of MessagePad. Also note that adding llamas to MessagePad will theoretically cause a multidimensional implosion. ("Don't cross the llamas, er . . . beams." — LlamaBusters)

Q *I've noticed some peculiar behavior in the Compile function and am wondering if it might be a bug. The problem is with special characters and string objects. When Compile is passed a string object containing special characters rather than a literal string with Unicode codes, the result is incorrect. This example works as expected:*

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Personal Interactive Electronics (PIE) division. Send your Newton-related questions to NewtonMail DRLLAMA or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt. •

```
x:= Compile("{msg: \"A string with special character \u00A5\u\"}";
y:= :x();
```
—> y is {msg: "A string with special character ¥"}

This example doesn't work as expected:

```
a:= "A string with special character \u00A5\u";
x:= Compile(a);
y:= :x();
```
*—> y is {msg: "A string with special character *"} where * is some character other than the expected "¥".*

Can you explain what's going on here?

A The problem is that you're using illegal NewtonScript syntax in the second example. If you used the inspector instead of Compile for this example, it would be like typing

```
A string with special character \u00A5\u
```

and then hitting Enter. This would result in a syntax error from NewtonScript. What you probably want is the equivalent of typing

```
"A string with special character \u00A5\u"
```

into the inspector. This is done with the following call to Compile:

```
x := Compile("\"A string with special character \\u00A5\\u\"");
call x with ();
```
—> #4415F49 "A string with special character ¥"

Note that the escape characters (\) for the Unicode string are themselves escaped. If you don't do this, you'll be putting the actual Unicode characters into the string being compiled, which is probably not what you want. Although your first example worked, you could easily get a case where not escaping the escape characters could bite you.

Q *In the communications input spec below, why does the call to UpdateStatus fail? UpdateStatus is a method in my base view, and the whole endpoint is in my base view, so why can't the input spec find the method?*

```
GetMessage: {
   inputForm: 'string,
   endCharacter: unicodeCR,

   InputScript: func(endpoint, data)
   begin
      :UpdateStatus(data);
      endpoint:SetInputSpec(GetMessage);
   end;
}
```

A The call to UpdateStatus fails because it's a message send that uses full inheritance to find the method. That means the system will look in the current

context (that is, self), then check the proto chain, and then check the parent chain. However, the current context is not what you think it is. In an input spec, the current context is the frame that defines the input spec. In this case, it's the GetMessage frame you define.

Since the GetMessage frame has no proto or parent pointer, the message send fails. There's a second problem waiting to happen: the call to SetInputSpec will also fail, because the symbol GetMessage isn't valid in this context.

The solution is to get a reference to your base view (or another view that contains or inherits the UpdateStatus message). The usual way to do this is to add an _parent slot to your endpoint at run time during initialization. Now your InputScript can use endpoint._parent to find the base view, as follows:

```
InputScript: func(endpoint, data)
begin
   endpoint:UpdateStatus(data);
   endpoint:SetInputSpec(endpoint.GetMessage);
end;
```

If you really want to use a simple message send (for example, :UpdateStatus), you could add an _parent slot to the input spec. This may be useful in situations where you have several input scripts that rely on a dynamic inheritance mechanism. That is, you change what the _parent slot of the input spec points to on the fly.

Q *Did you know that "gullible" is not in the Newton dictionary?*

A It is now.

Q *I have a large amount of static data in my application. I'd like to use Project Data to edit this data, but it won't fit. What can I do?*

A You must have an old version of the Newton Toolkit. As of version 1.0.1, the 32K limit is gone. You could use another text editor to edit the Project Data file. You could also use the Load command to load another NewtonScript source file.

As an example, assume you had a file called MyData.f in the same directory as your project and that this file contained the script that defined your constant data structures. You could use the Load command like this:

```
// This line appears in your Project Data file.
// Load in the data file and use the HOME compile-time variable
// to get the path to the project folder.
Load(HOME & "MyData.f");
```

Q *How can I figure out how much space my package and data will take on a card? I really want my application to fit on a 1-meg card.*

A The short answer is, you can't. The long answer is, load your packages and soups after completely erasing the card. To completely erase the card, open up preferences and then insert the card. Before the card is loaded, you'll get a chance to erase it.

Look at the difference in the free space on the card. Use the value in the card dialog. The value in the remove-package picker is the uncompressed size. You must erase the card before you check the free space difference.

Q *I have an input spec that receives data and places it into a queue. When I get data, I set a flag in my base view (DataInQ) that indicates data is available. I know the data is getting sent, but my input specs never seem to get called. What's going on?*

A The chances are that your base view has some code like this:

```
myBase.WaitForData := func()
   while Not DataInQ do nil;
```

You may have more statements in the loop, and you may be using **repeat** instead of **while**, but you probably have a loop that waits for the DataInQ flag to be set. The problem is that you're not giving control back to the NewtonScript thread so that it can process the pending InputScript call (from your input spec).

If you really need to wait for data, you can use either an idle script or a repeating delayed action. The idle script will be significantly easier to implement. You should make the delay on your idle script long enough to give time to the Newton. Also note that the Newton is a battery-powered device, and excessive use of this kind of programming tends to drain the users — I mean, batteries.

Q *I have an array of text elements called MyFirstArray in my Project Data file. I want to set the text of a clParagraphView that I open to an item in this array. The clParagraphView has a slot (strRef) that references MyFirstArray[0]. The first element appears as the clParagraph's view. There are four buttons on the base view, and depending on which button is tapped I want a different element of this array to be the clParagraph's text. When I try replacing MyFirstArray[0] in strRef during the viewSetupFormScript, I get as text "MyFirstArray[1]", not the text this represents. Here's the code in SetupFormScript in the clParagraph:*

```
SetValue(self, 'strRef, "MyFirstArray["&tempslot&"]");
```

tempslot is a slot in the base view where I store a value depending on which button is tapped. What's the problem?

A The basic answer is that your SetValue statement is incorrect. This statement sets strRef to the string "MyFirstArray[" concatenated with the string representation of tempslot concatenated with "]". What you really want is the string that's in MyFirstArray at the position defined by tempslot; that statement would be

```
SetValue(self, 'strRef, MyFirstArray[tempslot]);
```

But there are better ways to do this. Which method you use depends on when you set the text of the clParagraphView. If you set up things at open time, use the viewSetupFormScript, but just assign directly to the text slot:

```
clParagraphView.viewSetupFormScript := func()
   text := MyFirstArray[tempslot];
```

Remember that SetValue will also dirty the view and call RefreshViews. This isn't something you want to happen when you Open a view.

The other case is that the clParagraphView is already open. In this case, you can use a SetValue statement to set the text slot directly, instead of setting a strRef slot.

One other note: If the user can edit the strings you place in a clParagraphView, you must Clone the string. Otherwise you can get a "tried to modify read only object" error.

Q *How long does it take to train a llama to be a competent NewtonScript programmer?*

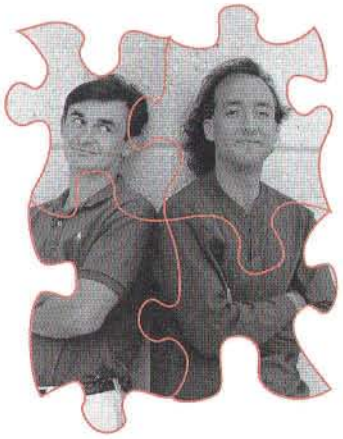
A About four weeks, but the hooves get in the way of really fast coding.

Thanks to our PIE Partners for the questions used in this column, and to jXopher, Todd Courtois, Bob Ebert, Mike Engber, Kent Sandvik, Jim Schram, Maurice Sharp, and Scott ("Zz") Zimmerman for the answers.

Have more questions? Need more answers? Take a look at PIE Developer Info on AppleLink.

Heaps of Fun

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL) — and a special guest, developer Steve Newman. The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.



KONSTANTIN OTHMER, BRUCE LEAK, AND STEVE NEWMAN

Steve — I've got a machine that crashed into MacsBug. I think it's this bug that some of our beta testers have been reporting; it's *really* intermittent, so I may not get it to happen again. I've got to find it just by looking at this one crash.

KON — It's not reproducible?

Steve — Not if it's the bug I've been hearing about. The reports are always the same: The machine crashes while saving a file. Afterward the file is unreadable. If they go back to an older copy of the file, the problem doesn't recur. No single user seems to have had this crash happen more than twice, and no one has been able to associate it with something they were doing in the program before they told it to save.

BAL — What does this program do?

Steve — It's a PIM — personal information manager. Data entry and dialog boxes and stuff. It's a pretty big program, but very vanilla in its use of the ROM — strictly Volume I stuff, plus the Memory Manager and File Manager, of course.

KON — You've tried stress testing? Heap scramble, low-memory conditions, MemHell, QC, all of that?

KONSTANTIN OTHMER AND BRUCE LEAK have given up sleep because they need all the time they can get to manipulate the penny stock market via the budding information superhighway. They're no longer trying to break the sound barrier, but are working on the Hedgehog barrier. BAL wonders, "What's that blue Hedgehog got that our green Armadillo doesn't have?"•

STEVE NEWMAN (AppleLink STEVENEWMAN) has been programming on the Macintosh since 1984. Currently, he works at Common Knowledge, Inc., writing information management tools. In a previous life he cowrote FullPaint, FullWrite Professional, and Spectre. When asked if he thought the Power Macintosh was the hottest new game platform he'd seen since the Atari 800, he replied "yes."•

Steve Yeah. It was a war zone, and we couldn't bring out the bug. But it just happened to one of our tech support people, Stephanie. I've taken over her machine until I can figure out what's going on. She closed a file, it asked her if she wanted to save changes, she clicked Yes, and it crashed into MacsBug with an illegal instruction.

BAL Illegal instruction? Sounds like you've branched off into the middle of nowhere. Where's the program counter?

100 Steve **wh pc** says we're in CODE segment 44, $017C bytes into a routine called Preflush. According to a link map I can look at on another machine, segment 44 has the file-saving code.

KON Is the heap trashed?

95 Steve MacsBug says the heap is fine.

BAL Perhaps some random memory-trashing bug has overwritten part of the code segment. Disassemble around the program counter.

90 Steve It looks like valid code, but the PC is in the middle of an instruction.

KON Do you have any purgeable code segments?

85 Steve We have a fairly complicated code segment management scheme based on reference counting. We're pretty careful about it, though; it's been a long time since we've had any problems there. As it happens, segment 44 is purgeable, but it has too many entry points to do reference counting, so we just unload it from our event loop.

BAL Sounds like code right out of the Finder. Let's try to find out how we managed to branch into the middle of an instruction. Do a stack crawl and see where we came from. Let's look at all the registers to see if one of them contains a clue as to how we got here.

80 Steve OK. **sc6** says the last call came from a function named "Document::SaveAs(int, unsigned char)".

KON What kind of a function name is that? It looks more like a UNIX pathname than a function name.

Steve Hey, you should see what it looks like with name unmangling disabled. **sc7** shows another return address under that one, in "Document::SaveAs(char*, short, unsigned char, int, unsigned char)".

BAL SaveAs? I thought it was just doing a regular save. And why two functions called SaveAs?

75 Steve Stephanie insists that it was a regular save — the document had been opened from an existing file and was being saved to that same file. But if that were true, the program should have called a function named Save, not SaveAs. As far as having two functions with the same name, the five-parameter one saves into a specified disk file; the two-parameter one brings up a Standard File dialog and then calls the five-parameter one. I'm using C++ function overloading.

BAL You sure you're not running the Finder? Mercer should be able to solve this for you in a snap.

KON I heard Mercer moved to Chicago. So, how did we get called?

70 Steve The call came from a "JSR (A1)" instruction. It looks like a standard C++ virtual function call.

KON What's the value of A1?

Steve It points into the jump table. Disassembling at that address shows a JMP to the current program counter.

KON That makes sense. Virtual function tables are stored in the global data segment, so their function pointers are data-to-code references, which have to go through the jump table. So all virtual function calls go through the jump table. That would be necessary no matter how the vtables were implemented, since at link time there's no way of knowing what version of the function will get called or what segment it's in.

BAL So the jump table is trashed relative to the data in the heap. I still think something's wrong with the heap. MacsBug can be funny about deciding whether a heap is trashed. Do a heap dump and page down until you see the block containing the program counter.

65 Steve It's code segment 44, and everything around it looks reasonable. But there's a question mark next to the master pointer address.

KON That probably means the master pointer doesn't point back to this heap block. Let's look at the header for the heap block and find its master pointer to double-check MacsBug. The format of the header depends on whether the machine is using 24- or 32-bit addressing. We can tell the current mode, which is probably the machine's standard mode unless we're in some slimy QuickDraw code, by looking at MacBug's status display along the left side of the screen.

Steve MacsBug says the machine is in 24-bit mode. It's an old IIci with only 8 meg of memory.

BAL In that case, the block header is 8 bytes long. The first byte is a tag byte that indicates the type of block (free, pointer, or handle) and the slop factor; the next three bytes are the size; and for handles, the final four bytes are the offset from the beginning of the heap zone to the master pointer for this block. Check that offset in the heap and make sure there's a valid master pointer there.

60 Steve It agrees with the location printed in the heap dump. But the value in that master pointer doesn't point back to this heap block. It turns out MacsBug won't flag this as heap corruption, but it will put a question mark next to the master pointer for blocks where the master pointer doesn't make sense.

KON Do a heap dump and keep paging down until you find the address that it does point to.

55 Steve It points to a block labeled "CODE segment 44". There are two code segment 44s in the heap!

KON Is there a question mark on this one?

50 Steve No, MacsBug seems to be happy with the second block. According to MacsBug, it has the same master pointer address as the first block. Both blocks are marked as being locked and purgeable.

BAL But the master pointer really does point to this block, so there's no question mark. And "locked and purgeable" is the expected state for a purgeable code segment that's currently loaded — the lock flag overrides the purgeable flag.

KON To decide whether a heap block is a resource, MacsBug looks at the resource flag for the block. In a 24-bit heap, that flag is stored in the high byte of the master pointer. Since both blocks think they have the same master pointer, they're sharing the same flag byte; and when

MacsBug searches the open resource maps to figure out which resource each block comes from, it gets the same answer.

BAL Now we have two mysteries: why there are two heap blocks with the same master pointer, and why the jump table points into the middle of a routine. Let's see if the heap blocks are really the same. Check the heap dump to see if they have the same size, and then dump memory from each one to see if they're the same.

45 Steve They are the same. And by the way, you left out one mystery: If we're doing a Save, why is SaveAs on the stack? There's no way that the two-parameter SaveAs can call the five-parameter SaveAs without first bringing up the Standard File dialog; but Stephanie insists there was no such dialog.

KON Maybe we took another bad branch through the jump table earlier on. Take another look at the stack crawl. When did we first enter segment 44?

40 Steve A routine called OKToClose, which is not in segment 44, called the two-parameter SaveAs, which is.

BAL Look at the JSR instruction in OKToClose.

35 Steve It's jumping to an A5-relative address, in the jump table. That address contains a JMP into the middle of the two-parameter SaveAs, shortly before the place where it calls the five-parameter version.

KON Aha! By taking a wild branch into the middle of the routine, it skipped over the call to Standard File. Maybe all the jump table entries for this segment are skewed by the same amount. Disassemble from $017C bytes above where this JMP points.

30 Steve $017C bytes above the JMP target is the beginning of Document::Save.

BAL That makes sense. OKToClose tried to call into segment 44 to the Save routine, but something went wrong with LoadSeg, and it ended up $017C bytes farther down, in the middle of the two-parameter SaveAs. Two-parameter SaveAs called five-parameter SaveAs; this is an intra-segment call, so it wasn't affected by the bad jump table. Then five-parameter SaveAs called Preflush, which is a virtual function, so it went through the jump table even though it's in the same segment. This time the wild branch happened to hit an illegal instruction, so it dropped into MacsBug.

KON It's interesting that the two SaveAs routines were able to function more or less correctly even though OKToClose branched into the middle of the first routine, thus bypassing all of its parameter setup.

BAL Well, it sounds like all of these functions are methods of the same object. MPW's C++ compiler usually puts the object pointer in A4. So any references to object data members or virtual functions would work even though we skipped the entry code for the first SaveAs.

KON Aren't all C++ functions fairly interchangeable? Link, save A4, load A4, test a bit off A4, restore A4, unlink, rts? That's part of the efficiency.

BAL In any case, we need to find out what went wrong in the LoadSeg call. Maybe there's a clue on the stack. Dump memory for a few hundred bytes starting at the stack pointer.

25 Steve $0028 bytes after the stack pointer, you notice a funny value: $4080BD0A.

KON That's an address in ROM, probably a return address. Disassemble around that address.

20 Steve It's in LoadSeg, one instruction after a call to StripAddress. It looks like this:

```
Disassembling from 4080bce0
_LoadSeg
    +0000 4080BCE0   MOVEM.L    D0-D2/A0/A1,-(A7)
    +0004 4080BCE4   MOVE.L     D1,-(A7)
    +0006 4080BCE6   JSR        Dispatcher+00C6
    +000A 4080BCEA   MOVE.L     (A7)+,D1
    +000C 4080BCEC   MOVE.W     $0018(A7),D0
    +0010 4080BCF0   BSR.S      LoadSeg+007C
    +0012 4080BCF2   BEQ.S      LoadSeg+0076
    +0014 4080BCF4   HGetState
    +0016 4080BCF6   BTST       #$07,D0
    +001A 4080BCFA   BNE.S      LoadSeg+0026
    +001C 4080BCFC   TST.B      SegHiEnable
    +0020 4080BD00   BEQ.S      LoadSeg+0024
    +0022 4080BD02   MoveHHi
    +0024 4080BD04   HLock
    +0026 4080BD06   MOVE.L     (A0),D0
    +0028 4080BD08   StripAddress
    +002A 4080BD0A   MOVEA.L    D0,A0
    +002C 4080BD0C   MOVEA.L    A5,A1
    +002E 4080BD0E   ADDA.W     CurJTOffset,A1
    +0032 4080BD12   ADDA.W     (A0),A1
    +0034 4080BD14   CMPI.W     #$4EF9,$0002(A1)
    +003A 4080BD1A   BEQ.S      LoadSeg+005C
    +003C 4080BD1C   MOVE.W     $0002(A0),D0
    +0040 4080BD20   BEQ.S      LoadSeg+005C
    +0042 4080BD22   MOVE.W     $0018(A7),D1
    +0046 4080BD26   MOVEQ      #$00,D2
    +0048 4080BD28   MOVE.W     (A1)+,D2
    +004A 4080BD2A   MOVE.W     D1,-$0002(A1)
    +004E 4080BD2E   MOVE.W     #$4EF9,(A1)+
    +0052 4080BD32   PEA        $04(A0,D2.L)
    +0056 4080BD36   MOVE.L     (A7)+,(A1)+
    +0058 4080BD38   SUBQ.W     #$1,D0
    +005A 4080BD3A   BNE.S      LoadSeg+0048
    +005C 4080BD3C   MOVEA.L    $0014(A7),A1
    +0060 4080BD40   SUBQ.L     #$6,A1
    +0062 4080BD42   MOVE.L     A1,$0016(A7)
    +0066 4080BD46   MOVEM.L    (A7)+,D0-D2/A0/A1
    +006A 4080BD4A   ADDQ.W     #$2,A7
    +006C 4080BD4C   TST.B      LoadTrap
    +0070 4080BD50   BEQ.S      LoadSeg+0074
    +0072 4080BD52   Debugger
    +0074 4080BD54   RTS
    +0076 4080BD56   MOVEQ #    $0F,D0
    +0078 4080BD58   SysError
    +007A 4080BD5A   Debugger
    +007C 4080BD5C   ST         ResLoad
```

```
+0080  4080BD60   SUBQ.W     #$4,A7
+0082  4080BD62   MOVE.L     #$434F4445,-(A7)    ;'CODE'
+0088  4080BD68   MOVE.W     D0,-(A7)
+008A  4080BD6A   GetResource
+008C  4080BD6C   MOVEA.L    (A7)+,A0
+008E  4080BD6E   MOVE.L     A0,D0
+0090  4080BD70   RTS
```

BAL The JSR to Dispatcher+00C6 flushes the instruction cache. Because the 68030 has separate instruction and data caches, LoadSeg needs to do that to make sure that the newly loaded data is eligible to make it into the cache. Next the subroutine at +007C gets the code resource. If the handle isn't locked, it's moved high and locked. Then we find the first jump table entry for the segment, and test to see if it's loaded by checking whether the first instruction is $4EF9 (a JMP.L). If it's not loaded, each entry for this segment is updated from the unloaded form (involving a call to LoadSeg) to the loaded form (involving a JMP.L). But it must have skipped this, because otherwise the PEA at +0052 would have overwritten the return address from the call to StripAddress, and that return address is still on the stack.

KON It skipped over the code to transform the jump table entries, so the segment must have already been loaded. But if the segment was loaded, the jump table wouldn't have any LoadSeg calls for that segment. Somehow LoadSeg was called for a segment that was already loaded. So your application must be calling LoadSeg manually!

15 Steve Honest, I'm not calling LoadSeg manually. A search of my source code verifies this.

BAL The only other way for LoadSeg to get called is through the jump table. How does your reference-counting segment unloader work? Is it possible that a segment gets called by your reference-counting code while you're in the process of loading it?

10 Steve It shouldn't be. The reference counting is done manually; we don't patch LoadSeg or anything nasty like that. At any rate, segment 44 isn't reference-counted.

KON Here's an idea: When LoadSeg was called to bring in segment 44, it called GetResource to bring the resource into memory. Assuming the code segment had been loaded in the past and later unloaded and purged, GetResource would have called ReallocHandle, which was short on memory and called your GrowZone hook. Your GrowZone function started freeing memory and then called another function in segment 44, triggering a recursive call to LoadSeg.

BAL With enough memory free, segment 44 was loaded. Then the GrowZone function exited back to the ReallocHandle call, which succeeded, and segment 44 was loaded again when the GetResource call completed. When the original LoadSeg checked the state of the jump table, it was already kosher, so the test for $4EF9 fired and the return address from StripAddress didn't get overwritten.

KON That certainly explains the confused heap. The GZSaveHnd that was passed to the GrowZone function shouldn't be touched, but you called GetResource on it indirectly via LoadSeg. It also explains the skewed jump table entries: after allocating a memory block, ReallocHandle simply assigns the master pointer to point to that block, without preserving the handle state stored in the high byte of the master

pointer. This effectively sets the handle state to 0, erasing the HLock call from the inner LoadSeg. Thus, when the outer LoadSeg called MoveHHi on the second copy of the segment, the lock bit in the master pointer — which is shared by both blocks — was clear. So when MoveHHi called CompactMem, the first copy of the segment was free to move (in this case, by $017C bytes). Finally, GetResource returned to the original LoadSeg, which set the lock bit again.

BAL Take another look at your link map. Are there any routines in segment 44 that could be called from your GrowZone hook?

5 Steve That's funny. There are some routines in this segment that shouldn't be there — in fact, they shouldn't be anywhere. They're supposed to be inline functions!

BAL The C++ compiler won't always copy a function inline, even if it's declared that way. This can happen if the function body is too complicated. Segment loading is a foreign concept that doesn't fit well into a C++ class hierarchy, and the MPW implementation has a few puzzlers.

KON Some of the calls to these "inline" functions were from segment 44, so they happened to be placed in that segment. Then, when the GrowZone hook tried to call one of the inline functions, it had to load segment 44 — and the rest is history.

BAL C++ claims another victim.

Steve So how do I avoid this in the future? Put segment #pragmas around all my inline functions?

BAL That's a superstition believed by some people who should know better. It doesn't work.

KON What does work is what those Finder folks did. MPW CFront puts the uninlineable functions at the end of the file it's compiling. The Finder folks just end every file with "#pragma segment CFrontCruft," and all the unexpected functions wind up in one easy-to-manage segment.

Steve "Uninlineable" isn't a word.

BAL That's why it's called cruft. Incidentally, this technique also catches functions that the compiler has to synthesize entirely, such as constructors and destructors for classes where they're needed (to initialize the vtable, for example) but aren't declared explicitly. And by looking at the link map, you can see what the compiler is doing behind your back — although you might be happier not knowing.

KON Nasty.

BAL Yeah.

SCORING

80–100 You should be a guest puzzler yourself; send in a draft to AppleLink DEVELOP.

55–75 Pretty sharp; maybe you can write the first hot OpenDoc container app.

30–50 Maybe you can write an OpenDoc part.

5–25 Maybe you'd better stick to AppleScript. •

Thanks to scott douglass for reviewing this column, and to Ludis Langens for wading into a haystack of hex and emerging with a needle labeled 4080BD0A. •

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. APDA, Apple's source for developer tools, and Apple Developer University are open to anyone who wants access to development tools and instruction. Developers may access additional information and services through Apple's Developer Programs.

APDA offers convenient worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers periodically receive the *APDA Tools Catalog* featuring hundreds of Apple and third-party development products. There are no membership fees. APDA offers convenient payment and shipping options, including site licensing.

Apple Developer University (DU) provides training designed to increase your software development productivity. The curriculum includes courses to get you started programming on Apple platforms, as well as advanced, in-depth training on the newest Apple technologies, such as PowerPC, OpenDoc, QuickDraw GX, and Newton. DU offers courses in Cupertino CA and at selected training locations. The DU Extension partner located in Portsmouth NH also schedules selected courses in its facilities.

In addition to classroom training, DU offers multimedia self-paced courses and low-cost mini-course tutorials.

The Associates Program is Apple's primary program for developers across all Apple technologies, including Macintosh, Personal Interactive Electronics (such as Newton), and multimedia. It's a low-cost, self-support program that also provides a connection with Apple and fellow developers, information on new technologies, and discounts on equipment.

The Apple Multimedia Program is designed for developers interested in the emerging multimedia market. Program features include a quarterly mailing and discounts on third-party products, training, and events.

The Macintosh Technology Partners Program is open to Apple-selected strategic developers focused on Macintosh technology, including PowerPC, QuickTime, QuickDraw GX, and PowerTalk. In addition to receiving the same development information and tools as members of the Associates Program, Macintosh Technology Partners receive programming-level development support via electronic mail. Membership in this program is limited to strategic developers who directly contribute to Apple's long-term product plans and business objectives.

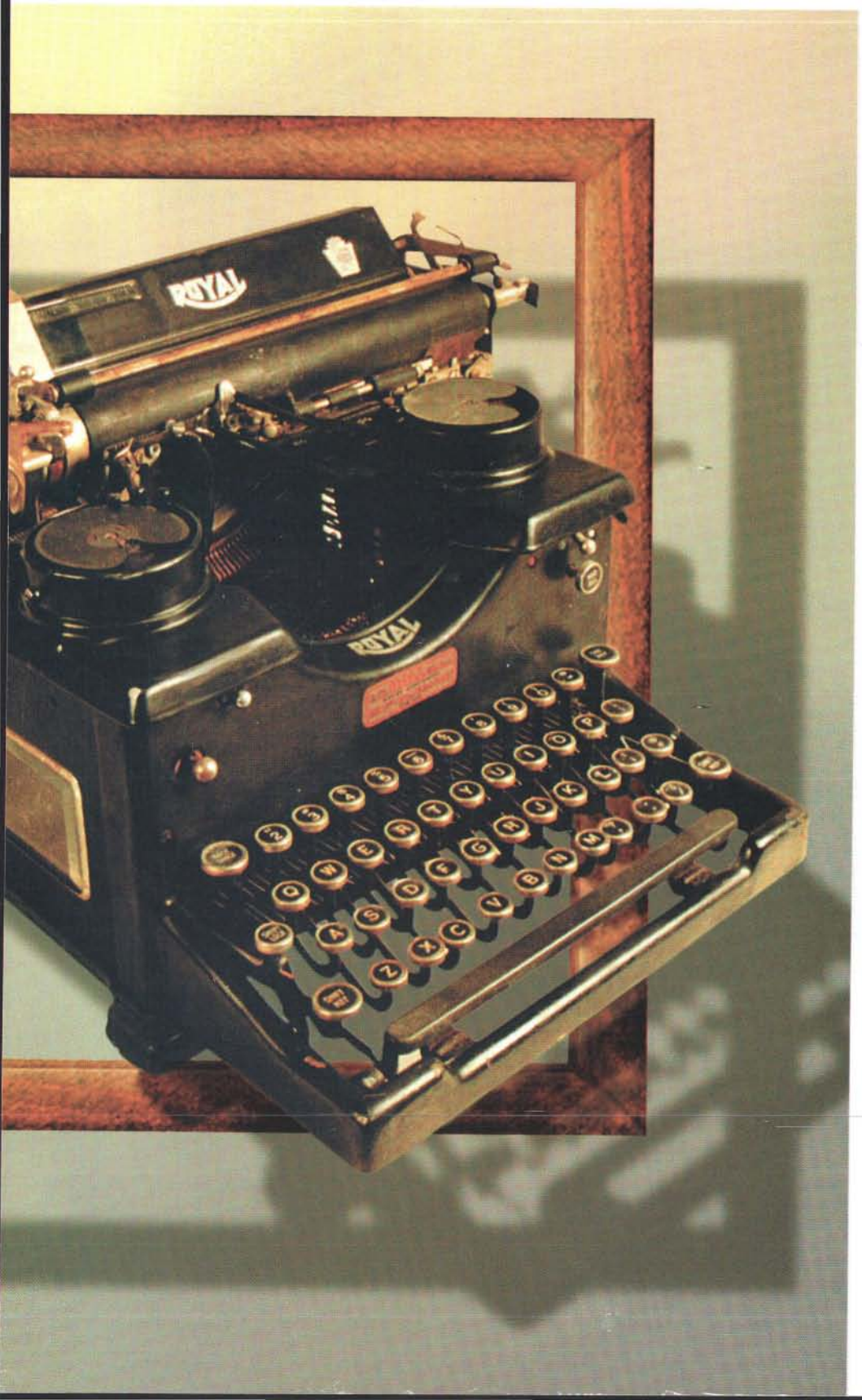
The PIE Partners Program is open to Apple-selected strategic developers focused on Personal Interactive Electronics. It offers the same core features as the Associates Program, but also includes programming-level development support via electronic mail, additional hardware purchasing privileges, marketing programs, and media production assistance.

APDA To order products or receive a complimentary catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also order electronically (AppleLink APDA; Internet apda@applelink.apple.com; America Online APDAorder; or CompuServe 76666,2405) or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

Apple Developer University The registrar at (408)974-4897 can reserve your place or send a current Curriculum Guide and Course Schedule. You can also send an AppleLink to DEVUNIV or write Developer University, Apple Computer, Inc., One Infinite Loop, M/S 305-1TU, Cupertino, CA 95014. Self-paced products should be ordered directly through APDA.

Apple Developer Programs Call the Developer Support Center at (408)974-4897, AppleLink DEVSUPPORT, or write One Infinite Loop, M/S 303-2T, Cupertino, CA 95014, for information or an application form. Developers outside the U.S. and Canada should instead contact the Apple office in their country for information about developer programs.

19



Apple Computer, Inc.
One Infinite Loop
Cupertino, CA 95014

BULK RATE
U.S. POSTAGE
PAID
SAN BERNARDINO, CA
PERMIT #448